THE COMPLEXITY OF SPEED-SCALING

by

Neal Barcelo

Bachelor of Science, Denison University, 2011

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences in partial

fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2015

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Neal Barcelo

It was defended on

April 29th 2015

and approved by

Kirk Pruhs, Department of Computer Science

Anupam Gupta, Department of Computer Science

Adam Lee, Department of Computer Science

Daniel Mossé, Department of Computer Science

Dissertation Director: Kirk Pruhs, Department of Computer Science

THE COMPLEXITY OF SPEED-SCALING

Neal Barcelo, PhD

University of Pittsburgh, 2015

It seems to be a corollary to the laws of physics that, all else being equal, higher performance devices are necessarily less energy efficient than lower performance devices. Conceptually there is an energy efficiency spectrum of design points, with high performance and low energy efficiency on one end, and low performance and high energy efficiency on the other end. As in most technologies, there is not a universal sweet spot for computer chips that is best for all situations. For example, there are situations when there are critical tasks to be done, when performance is more important than energy efficiency, and there are situations when there are no critical tasks, when energy efficiency may be more important than performance. Speed-scalable processors aim to resolve this conflict by allowing the operating system to dynamically choose the performance to energy trade-off. The resulting optimization problems involve scheduling jobs on such a processor and have conflicting dual objectives of quality of service and some energy related objective. This engenders many different optimization problems, depending on how one models the processor, the performance objective, and how one handles the dual objectives.

In this thesis we map out a reasonably full landscape of all possible formulations, determining which assumptions drive computational tractability. Beyond identifying the individual computational complexities, we use algorithmics as a lens to study the combinatorial structure of such trade-off schedules. In particular, we give several general reductions which, in some sense, reduce the number of problems that are distinct in a complexity theoretic sense. We show that some problems, for which there are efficient algorithms on a fixed speed processor, are NP-hard. Finally, we present several primal-dual based polynomial time algorithms.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

PREFACE

First and foremost I would like to thank my advisor Kirk Pruhs for four years of guidance, encouragement, and invaluable advice. This thesis would not be possible without his dedication as an advisor.

I would like to thank my numerous coauthors, Antonios Antoniadis, Daniel Cole, Mario Consuegra, Kyle Fox, Peter Kling, Sungjin Im, Ben Moseley, Michael Nugent, and Michele Scquizzato, who made much of this research possible. I would also like to thank my committee members, Anupam Gupta, Adam Lee, and Daniel Mossé, for taking time out of their busy schedules to serve on my committee.

A special thanks to all my friends and family for their continued support, especially E, who is always pushing me to challenge myself and always there to listen.

Lastly I would like to thank Jenn for being a constant source of support and inspiration throughout these four years.

1.0 INTRODUCTION

It seems to be a corollary to the laws of physics that, all else being equal, higher performance technology is necessarily less energy efficient than lower performance technology. For example, a Ferrari sports car is less energy efficient than a Toyota Prius, and it is not possible with current technology to get Ferrari performance with Prius fuel efficiency. Conceptually there is an energy efficiency spectrum of design points, with high performance and low energy efficiency on one end, and low performance and high energy efficiency on the other end. Early in the design process a car designer has to select a sweet spot on this spectrum as a design goal. There is seldom a uniquely defined sweet spot that is best for all situations, which is why both sports cars and economy cars are produced.

Although information technology is no exception to this corollary, unlike other technologies, historically the sweet spot was invariably skewed towards the high performance low energy efficiency end of the spectrum. Energy consumption was simply not a first order concern and computer chip designers competed primarily on performance. But about a decade ago, due in large part to the exponentiality of Moore's law and associated thermal and power implications, there was a relatively abrupt shift in the design sweet spot for major chip manufacturers towards the middle of the energy efficiency spectrum. Transistor densities had been exponentially scaling up for over a decade and chip manufacturers could no longer scale down the power per transistor fast enough to keep power density constant. This, combined with the fact that allowing power density to further increase was infeasible from a thermal standpoint brought energy to the ranks of space and time as a scarce computational resource.

With energy consumption becoming a first order concern, there was no longer a universal sweet spot for computer chips that was best for all situations, which still holds true today. There are situations, for example when there are critical tasks to be done, when performance is more important than energy efficiency, and there are situations, for example when there are no critical tasks to be done, when energy efficiency may be more important than performance. As such, if forcing the designer to choose the sweet spot during the design process, it is unavoidable that this choice will be suboptimal for certain workloads. However, unlike for cars, it is technologically and economically feasible to design computer chips with multiple operational modes, each at a different point on the performance and energy efficiency spectrum. So instead of statically determining the sweet spot during the design process, the decision is passed on to the operating system to be made dynamically depending on current demands.

1.0.1 Speed-Scalable Processors

The most prevalent manifestation of this technology is a speed-scalable processor, as manufactured by the likes of Intel and AMD, which enables the operating system to control the speed of the processor dynamically. In practice these processors have a discrete number of modes, where each mode has a unique speed and power consumption, with the higher speed modes being less energy-efficient in that they consume more energy per unit of computation. This is achieved through a technique called dynamic voltage and frequency scaling (DVFS) where a decrease in CPU frequency allows for a corresponding decrease in voltage yielding potentially significant energy savings. Since the power is observed to be approximately the voltage squared, even modest frequency reductions (and therefore voltage reductions) can lead to orders of magnitude savings in energy. In practice the decision of which operational mode to occupy is made using crude approximations of the current workload of the system either at the operating system or hardware level. That is, when the system is idle the operating system will choose a low performance high energy efficiency state and a high performance state when there is significant workload. While such an algorithm undoubtedly offers some balance between performance and energy efficiency, a priori it is not clear it offers the optimal balance, however we choose to define this. In this thesis we rigorously explore how to achieve such an optimal balance for these conflicting objectives.

1.0.2 Research Objectives

As we will see, abstracting this optimization problem involves several modeling choices, the appropriateness of which will depend on the underlying workload as well as the system specifications. With this in mind, there are four primary research objectives of this thesis. One objective, is for each setting, to determine the offline computational complexity of the resulting optimization problem at the granularity of P vs. NP-hard. Here by offline we mean the algorithm has access to the entire input at the beginning of its execution. Resolving the complexity of each setting will enable practitioners to either utilize our algorithms, or in the case of hardness results, avoid attempting to solve the problem optimally. Given that each model is motivated by a realistically occurring system/workload, it is important we determine this distinction in complexity across all settings.

Our second objective is to examine the complexity and structural insights gleaned from this first objective on a broader scale through the use of the taxonomy presented in Table 2. So while our first objective is concerned with properties of individual settings, this objective takes a more holistic view examining the entire landscape of possible formulations. In doing so, we hope to isolate modeling decisions that are sufficient for determining complexity and drive structural changes. This broader context will enable theoreticians and practitioners alike to better reason about the implications of their modeling and design decisions.

Turning to our third objective, rather than simply finding an efficient algorithm to solve the problem at hand, we aim to use Algorithmics (that is, finding provably good algorithms) as a lens to study the combinatorial structure of optimal schedules. When designing (not necessarily provably good) heuristic algorithms, it is difficult to uncover and utilize optimal structure, as it is often complicated and unintuitive. However when searching for provably good algorithms we are forced to uncover such general structural properties if we want any hope of reasoning about an optimal schedule on an arbitrary input. Beyond enabling this analysis of our offline optimal algorithms, in both design and analysis of approximation and online algorithms, utilizing structural insights is often crucial to developing provably good algorithms. So by deepening our understanding of optimal structures, even in the case of hardness results, this may be useful for analyzing other, not necessarily optimal, algorithms.

Our fourth objective follows in a similar vein in that we hope to use Algorithmics to enable us to better reason about energy as a computational resource in general. More precisely, our fourth objective is to develop algorithmic techniques that will be useful in the analysis of a broader class of energy related optimization problems. Doing so will entail drawing parallels between the high-level algorithmic approaches for each individual setting. Our hope is these will prove useful as a starting point for the design of algorithms across other energy aware technologies.

1.1 MODELING DECISIONS

The resulting optimization problems involve scheduling jobs/tasks on a such a speed-scalable processor and have dual objectives of quality of service and some energy related objective. An algorithm, therefore, must specify at each time which task to assign to the processor and at what speed. This engenders many different optimization problems, depending on how one models the jobs, the processor, the performance objective, and how one handles the dual objectives.

There are two competing goals in capturing such an optimization problem for theoretical analysis. On the one hand, we want to capture reality as accurately as possible so that practitioners may utilize our algorithms and insights with minimal modifications. On the other hand, we must balance this with choosing models that are mathematically tractable, which often means abstracting away many of the underlying details. As we discuss our modeling decisions below, we make an effort to identify which end of this spectrum each decision is skewed towards.

- **Jobs:** We inherit the standard model of jobs/tasks from the Operating Systems and Scheduling literature. A job/task will consist of a release time, some amount of work, and (perhaps) a priority. Breaking each of these down, the release time represents the first time the processor can begin working on this task, that is, the time the job is submitted to the system. To reason about the workload of a job, think of a single unit of work representing an infinitesimally small instruction to be executed. Note this is not the same as the time required on the processor since we assume that the processor can operate at multiple speeds. Finally, the priority of a job will be utilized in capturing the performance objective and allows the input to specify relative importance between jobs. For example, as alluded to before, inputs in which there are critical (high priority) jobs as well as non-critical (low priority) jobs will likely benefit from power heterogeneous technologies.

- **Processor:** Modeling the processor involves deciding the set of allowable speeds, as well deciding any restrictions on the speed to power function. In terms of allowable speeds, perhaps best modeling reality, in the discrete setting we assume that the processor consists of a discrete set speeds. While this model is most realistic, it is often more mathematically convenient to assume the processor can run at any nonnegative speed. As a compromise, we can also consider the model where the processor can run at any nonnegative speed less than some bounded maximum speed.

  In modeling the speed to power function, in the discrete setting we assume each discrete speed has an associated discrete power. For the two continuous settings, there is some function P that maps a speed s to a power $P(s)$. Most commonly one assumes $P(s) = s^{\alpha}$ for some constant α, slightly generalizing the well-known cube-root rule that the speed is approximately the cube-root of the dynamic power. We will also consider slightly more general power functions that obey some "nice" properties (i.e. $P(s)$ is convex).

- **Performance Objective:** There are two commonly used methods of enforcing quality of service within the scheduling literature. The first is to impose deadlines on jobs, in some sense turning the performance objective into a constraint. For example imagine a sensor which must collect and report data on some regimented schedule. Each collection would represent a task and the the collection schedule would induce a set of deadlines.

  However, in many cases, it is either unnatural or infeasible to formulate hard deadlines for each task. In this case we turn to our second performance metric, flow/waiting time. At a high level, flow/waiting measures how long a job/work is in the system. There are two sub classification depending on whether we calculate the waiting time of each unit of work, or the waiting time of the an entire job. In the fractional flow setting, we assume that each infinitesimal unit of work has its own flow time, and the fractional flow of a job is the sum of the flow of each unit of work. This is most appropriate when the user will benefit from partial completions. This contrasts to the integral flow setting where the flow time is the completion time of a job minus the release time of a job. To better understand this distinction, consider the context where jobs are flight queries to a travel site. Aggregating over the delay of jobs is more appropriate in the context of Orbitz, as Orbitz does not present the querier with any information until all possible flights are available, while in the case of Kayak, aggregating over the delay of work may be more appropriate as Kayak presents the querier with flight options as they are found.

Lastly, note that in order to incorporate the priority/weight of a job into the performance objective, we may consider the weighted flow time of job/work, which we calculate by multiplying the weight of a job by its flow time.

- **Energy Source:** In many speed scaling problems, power source assumptions are not explicitly stated, however, as we will see in the final modeling decision, examining the objective or constraints implicitly determines the most appropriate power source. In the first setting, we assume the system consists of a battery with some finite stored energy, limiting the total amount of energy the system can use. As a slight variation, we also consider the setting where the system is equipped with technology to both harvest and store energy from its environment. For example, a sensor that is equipped with a solar cell and a battery. Finally, our last power source model assumes the system has an unlimited energy supply essentially removing any energy related constraints. As we will see below, the power source is intimately tied to how we deal with the dual objectives.
- **Dual Objectives:** Finally, we must make a modeling decision as to how we combine the quality of service and energy objectives. As noted above, the dual objectives of performance and energy efficiency are conflicting in that increasing performance decreases energy efficiency and vice versa. When modeling an optimization problem with multiple objectives we must decide whether to combine these into a single objective (for example using a linear combination) or convert one into a constraint (for example limiting the amount of energy the schedule can use). The appropriateness will depend on the power source as well as whether knowledge regarding the desired energy performance trade-off is known.

		Unweighted Jobs		**W**eighted Jobs	
		Unit Sizes	**A**rbitrary Sizes	**U**nit Sizes	**A**rbitrary Sizes
Fractional Flow	**D**iscrete Speeds	? / ?	? / ?	? / ?	? / ?
	Continuous Speeds	? / ?	? / ?	? / ?	? / ?
Integral Flow	**D**iscrete Speeds	? / ?	? / ?	? / ?	NP-hard [2] / NP-hard [2]
	Continuous Speeds	P [AF 07] / P [PUW 08]	? / ?	? / ?	NP-hard [2] / NP-hard [2]

Table 1: Summary of previous results in the context of full range of possible settings. For each cell, the upper-half refers to the flow $+ \beta \cdot$ energy objective and the lower-half refers to flow minimization subject to an energy constraint.

Take the case where the system consists of a laptop powered by a battery. In this case, the optimization problem that minimizes the quality of service objective while ensuring the energy used is less than the total amount stored in the battery would be most appropriate. However when the power source is unlimited and the tradeoff between energy and performance is known a priori, it may make more sense to consider minimizing the sum of the quality of service and energy metrics. The two optimization problems are closely related, and foreshadowing slightly, our results will show that in many cases providing an efficient algorithm for one immediately yields an efficient algorithm for the other setting. Finally, note that in the case of deadline problems our quality of service objective is cast as a constraint and we will focus on minimizing some energy related objective.

1.2 PREVIOUS RESULTS

We provide here a survey of the most relevant related work, segmented according to the performance objective, flow time problems and deadline feasibility problems.

Flow Time Optimization Problems: Given the wide array of possible flow time optimization problems, we utilize the following succinct notation which captures the modeling decisions discussed in previous subsections. The format of our description is essentially a 5-tuple of the form `*-****`. The first entry captures the objective (**B**udget or **F**low plus **E**nergy). The remaining entries are **I**ntegral or **F**ractional flow, **C**ontinuous or **D**iscrete speed, **W**eighted or **U**nweighted, and **A**rbitrary or **U**nit size. A `*` represents a "don't care" entry. See Table 1 for an overview that puts all the flow time related results into the context of the full range of possible problems.

- `B-ICUU:` [31] gave a polynomial-time homotopic optimization algorithm for the problem of minimizing integral flow (I) with continuous speeds (C) subject to an energy budget (B) for unweighted jobs (U) of unit size (U) using the power function $P(s) = s^{\alpha}$. The first key insight was that given that the jobs were unweighted and unit size, the optimal schedule will complete jobs in First-In-First-Out order. Coupling this with the optimality conditions for the natural convex program allowed them to guid their algorithm in a homotopic fashion. Unfortunately for all cases outside of the unweighted unit size cases, this approach breaks down as the optimal completion ordering is not necessarily First-In-First-Out.
- `FE-ICUU:` [2] gave a polynomial-time dynamic programming algorithm for the problem of minimizing integral flow (I) with continuous speeds (C) for unweighted jobs (U) of unit size (U) and the objective of flow plus β energy (FE). So this was essentially the same as the previous setting, only changing the assumption on the power source from a battery to unlimited supply. Since they were still in the unweighted unit size case, they were again able to leverage the FIFO completion ordering. With this completion ordering they are able to compute the optimal schedule for a sequence of jobs that are "overlapping" (that is, $c_i > r_{i+1}$). Finally, a dynamic programming formulation which computes the portions of the schedule that should be overlapping yields their final algorithm.

- `*-I*WA`: There is a subset of hardness results that follow from hardness results in the classical single speed scheduling literature. Namely, [25] gave NP-hardness results for a fixed speed processor in the setting of integral flow (I), weighted jobs (W) of arbitrary size (A). The only observation required to extend these to the speed scaling setting is that one can tweak the speed to power function in such a way that the optimal algorithm will only utilize one speed. Note that these hardness results hold regardless of the assumption on discrete versus continuous speeds, as well as the performance objective assumption.
- `Approximation Results`: There is a large body of work concerning approximately computing optimal trade-off schedules, both offline and online. Since these are of little relevance to the work presented in this thesis we consider them in minimal detail. [28] gives PTAS's for minimizing total flow without release times subject to an energy budget in both the continuous and discrete speed settings. [2, 3, 7, 9, 10, 15, 18, 26] consider online algorithms for optimal integral flow and energy, whereas [7, 10, 18] consider online algorithms for fractional flow and energy. In particular, [10] show that there are $O(1)$-competitive algorithms for all of the flow plus β energy problems that we consider (with arbitrary power functions). For a survey on energy-efficient algorithms, see [1]. For a fixed speed processor, all the fractional problems can be solved by running the job with highest density (=weight/size). Turning to integral flow, if all jobs are unit size, then always running the job of highest weight is optimal. The complexity of the problem if all jobs have the same (not unit) size is open [11, 12]. The complexity of `FE-I*WU` seems at least as hard (but perhaps not much harder) than this problem. Finally, if all jobs have unit weight, then Shortest Remaining Processing Time is optimal for total flow.

Deadline Optimization Problems: The number of distinct settings in the deadline optimization space is far less as jobs do not have priorities (at least not explicitly), and there is no benefit to partially completing a task. That said, there are two additional subcategories we consider depending on whether the power source is a battery or a solar cell.

- `YDS`: In the seminal work of [34] the authors consider the problem of minimizing the total energy of a deadline feasible schedule where jobs have arbitrary release times, deadlines, and sizes. While not explicitly assumed, this is setting is most appropriate when the power source is a battery. They provide a simple polynomial time algorithm that is built on identifying intervals with the largest work to size ratio, scheduling the work in this interval and recursing. We will see this algorithm arise in Chapter 4 as we look at a similar deadline problem.

- `Solar Cell:` [8] were the first to consider a variation of this setting, in which the system harvests energy via a solar cell which can be stored in a battery. The objective was to find the minimum recharge rate and associated schedule such that each job completes before its deadline and the battery is never depleted. Here, the recharge rate is the rate at which energy is being harvested via the solar cell. So intuitively they are trying to produce the schedule that will be feasible in the most adverse weather conditions. While they do not give a combinatorial algorithm, they showed that the offline problem could be expressed as a convex program and thus in principle the problem is solvable in polynomial-time.
- `Approximation Results:` Again, there is a significant body of results concerned with approximately computing optimal schedules. In the solar cell setting, [8] proved that the schedule that optimizes the total energy usage is a 2-approximation for the objective of recharge rate. They also showed that the online algorithm BKP, which is known to be $O(1)$-competitive for total energy usage [6], is also $O(1)$-competitive with respect to the recharge rate. So, intuitively, the main take-away point from [8] was that schedules that naturally arise when considering the objective of energy usage are $O(1)$ approximate with respect to the objective of recharge rate.

  In regards to approximation results for the battery setting, there has been a stream of improvements upon the competitiveness of the initial online algorithms proposed in [34]. See [1] for a survey of the most recent results.

Real-Time Scheduling Contributions

There is a considerable amount of work from the real-time scheduling community that considers scheduling on speed scalable processors. All of the results discussed here assume that tasks are subject to hard deadline constraints and the objective is to minimize energy. Turning first to aperiodic tasks, [23] consider the setting of discrete speeds where all jobs arrive at time 0 and must be processed by some common deadline T, showing that the optimal algorithm will always utilize only two consecutive speeds, interpolating so that the last piece of work is processed at time T. Building on this, [24] combines the techniques of [23, 34] to develop an optimal algorithm for the setting of discrete speeds and arbitrary release times. [27] further improved on this setting, giving an algorithm with runtime of $O(kn \log n)$ where k is the number of discrete speeds. In addition to these optimal algorithms for aperiodic tasks, several papers consider heuristics with fixed priority scheduling algorithms motivated by the fact that fixed priority schedulers require less overhead than optimal scheduling policies such as EDF and are therefore often adopted in practical real-time schedulers [32, 33]. While less relevant to this work, there is also a considerable amount of work studying the periodic setting. In the continuous speed setting, [5] provide a heuristic which first considers the optimal worst-case speed and subsequently adjusts the speed online to "reclaim energy". Slightly generalizing this result, [4] remove the assumption that each task has identical power functions. For the discrete speed setting several approximation results are known [17, 29]. For a more comprehensive survey of the real-time scheduling communities contributions to energy aware scheduling see [16].

Queuing Theory Contributions

There has also been considerable contributions to energy aware scheduling and power management from the queuing theory community. Unlike the previous settings which consider worst-case input sequences, the work discussed here assumes inputs are defined by some probability distribution and the objective is to minimize in expectation some function of delay or performance. Perhaps most relevant to the work presented in this thesis is the model that allows for each server to have DVFS capabilities e.g. [14, 20]. Much like prior sections, the resulting optimization problems vary depending on whether energy is cast a constraint or objective. As a slight restriction on this model, there are several papers that consider the setting in which each server can occupy one of three states, on, off or idle e.g. [19, 21, 30]. This model is not subsumed by the DVFS model as they typically assume there is some start-up cost for switching a processor between the off and on states.

		Unweighted Jobs		**W**eighted Jobs	
		Unit Sizes	**A**rbitrary Sizes	**U**nit Sizes	**A**rbitrary Sizes
Fractional Flow	**D**iscrete Speeds	P [⋆] ≡ P [⋆]	P [⋆] ≡ P [⋆]	P [⋆] ≡ P [⋆]	P [⋆] ≡ P [⋆]
	Continuous Speeds	P [⋆] ≡ P [⋆]	P [⋆] ≡ P [⋆]	P [⋆] ≡ P [⋆]	P [⋆] ≡ P [⋆]
Integral Flow	**D**iscrete Speeds	P [⋆] ≡ P [⋆]	? NP-hard [⋆]	? NP-hard [⋆]	NP-hard [2] NP-hard [2]
	Continuous Speeds	P [AF 07] ≡ P [PUW 08]	? NP-hard [⋆]	? NP-hard [⋆]	NP-hard [2] NP-hard [2]

Table 2: Summary of our results in the context of the full range of possible settings. For each cell, the upper-half refers to the flow $+ \beta \cdot$ energy objective and the lower-half refers to flow minimization subject to an energy constraint. The symbol [$\star$] indicates results presented in this paper, and $\equiv$ indicates that two problems are computationally equivalent.

1.3 OUR CONTRIBUTIONS

In Chapters 2 and 3, we provide several results to more fully map out the landscape of complexity and algorithmic results for the range of flow time problems reflected in Table 2. In particular, for each setting our aim is to either give a polynomial time combinatorial algorithm (i.e., without the use of a convex program) or show it is NP-hard. In Chapter 4, we take a detour to the setting where quality of service is imposed via deadlines and the power source is a solar cell. We give a brief summary of the results for each Chapter as well as some high level intuition behind either the algorithms or hardness proofs.

1.3.1 Chapter 2

We begin in Chapter 2 with a polynomial-time algorithm for the problem of minimizing fractional flow (F) with discrete speeds (D) for weighted jobs (W) of arbitrary size (A) and

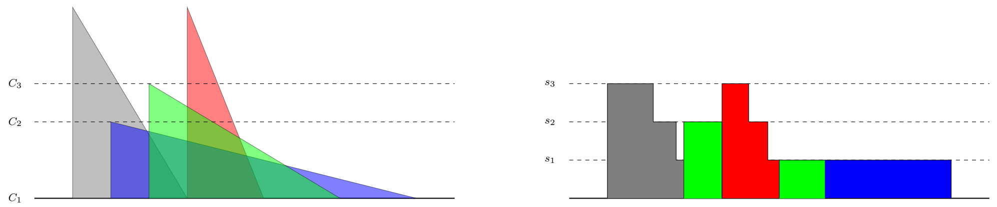


Figure 1: The dual lines for a 4-job instance and the associated schedule.

the objective of flow plus β energy (FE). Given the complexity of the resulting algorithm and analysis we devote an entire chapter to this problem.

The first step, and a recurring theme throughout this work, is to take the natural linear programming formulation of this optimization problem and consider the complimentary slackness conditions which characterize the structure of an optimal solution. The main insight from these optimal conditions is that by using a geometric interpretation of these conditions and creating a mapping between this interpretation and schedules, we reduce the original optimization problem to finding a set of "dual lines" whose corresponding schedule is feasible. Every job will have a "dual line" which will have two components. An initial value, which corresponds to the value of a dual variable, and a slope, which the optimality conditions tell us will always be $-p_j/w_j$ (where p_j is the size of job j and w_j is the weight of job j). The mapping between dual lines and schedules is then simply the upper envelope of the set of dual lines. So for all times t, whichever job has the highest dual line will be the job scheduled, and the speed will depend on the height of the dual line. See Figure 1 for an example of a set of dual lines and the corresponding scheduling. While these optimization problems are equivalent, the hope is that the geometric setting is easier to reason about.

We indeed show this to be true by giving a polynomial time algorithm for this geometric interpretation of the problem. The algorithm constructs an optimal schedule job by job, ensuring that as a new job is added, it maintains the optimality of the previously scheduled jobs. Taking the first job as an example, we will raise the dual line of this job until the corresponding schedule has completed the correct amount of work. As we raise the dual line of the second job, it may "steal away" work from the first job if the two dual lines intersect thereby changing the upper envelope for the first job. To compensate for this, we maintain an "affection tree" between jobs (roughly speaking a job j_1 "affects" j_2 if changing the "dual line" of j_1 impacts the resulting schedule of j_2) and simultaneously changing the dual line of a job and all jobs it affects. So in the case of our two job example, when the second dual line begins to intersect the first dual line, we will now simultaneously raise both dual lines at rates such that there is no change in the amount of work done by the first job. By continuing to do this for each job until the resulting schedule is feasible we produce an optimal schedule.

To bound the running time we bound the number of events that cause us to recalculate how we raise dual lines, the time required to calculate the next event of each type, and the time required to recompute the affection tree after each event. Intuitively, the events that cause us to recalculate are any events that change the affection tree, or change the rate at which work is changing for a job.

1.3.2 Chapter 3

In Chapter 3 we provide a survey of hardness results and polynomial time algorithms across a variety of settings to fill in Table 2. In addition, we give several reductions between different settings linking their complexity, thereby reducing the number of truly distinct problems. The results in this Chapter share the similarity that they follow from previous work or rely on rather standard techniques. Because of this, and the sheer volume of results, we give a compact summary and leave the technical explanations for the main body.

- Hardness Results
  - *`B-IDUA` is NP-hard:* The reduction is from the subset sum problem. The basic idea is to associate several high density and low density jobs with each number in the subset sum instance, and show that for certain parameter settings, there are only two possible choices for this set of jobs with the difference in energy consumption being this number. Perhaps the most interesting aspect of this result is not the technique, but the existence of a polynomial time algorithm for the fixed speed setting. As far as we know, this was the first known example of such a "hard" speed scaling problem with an "easy" fixed speed equivalent.

  - *`B-IDWU` is NP-hard:* The reduction is again from the subset sum problem but much more technically demanding than `B-IDUA`. Unlike the previous result, it is currently open whether the fixed speed equivalent is solvable in polynomial time.

- Polynomial Time Algorithms
  - *`FE-ICUU` is in P:* In our first positive result, we show that [2] can be extended to general power functions. Recall that [2] first realized the completion ordering must be FIFO and used this to solve the special case when a sequence of jobs are overlapping. We follow this same approach, noticing that set of equations arising from the special case can be solved for general "nice" power functions as opposed to simply $P(s) = s^\alpha$. From there a similar dynamic programming approach suffices.
  - *`FE-IDUU` is in P:* Again, the algorithm highly leverages the structure of the unit size unweighted case. The fact that speeds are discrete allows for a much simpler algorithm than for `FE-ICUU`. This theme of the discrete setting allowing for simpler algorithms will be recurring throughout.
  - *`FE-FCWA` is in P:* We generalize the algorithm from Chapter 3 to continuous speeds. Again, the main algorithmic ideas do not change, but instead the main hurdle is to deal with a more complicated equation system at certain points in the algorithm resulting from the generalized power function.
- Equivalence Reductions
  - *Reduction from `B-FC**` to `FE-FC**`:* We reduce any energy budget problem with fractional flow and continuous speeds to the corresponding flow plus β energy problem using binary search. So this tells us that providing an algorithm for the flow plus energy problem will immediately yield an algorithm for the equivalent energy budget problem.
  - *Reduction from `B-ICUU` to `FE-ICUU`:* The difficulty here stems from the fact that there may be multiple optimal flow plus energy schedules for a β (so binary search over β does not suffice). Again, this allows us to tie the complexity of the energy budget problem to the flow plus energy problem.
  - *Reduction from `*-*D**` to `*-*C**`:* We give a reduction from any discrete speed problem to the corresponding continuous speed problem. So this tells us that the

continuous version of a problem will always be harder, in a complexity sense, than the discrete version of a problem.

1.3.3 Chapter 4

In the final technical Chapter, we examine a speed-scaling optimization problem where the energy is harvested from a solar cell and quality of service is imposed via job deadlines as opposed to the flow/waiting time. This is most appropriate in the context of some sensor based device, many of which contain technologies enabling them to harvest energy from their environment thereby allowing them to be left unattended for long periods of time. The motivation of this Chapter is to consider, from an algorithmic perspective, how embedding speed scalable processors within such devices can extend their lifetime while simultaneously maintaining quality of service guarantees. In order to describe our results we first provide a brief reminder of the differences in modeling assumptions:

- The device harvests energy from its environment. For simplicity we will assume that it harvests energy at a time-invariant rate (like a solar-cell in bright sunlight).
- The device contains a battery that can store the harvested energy. For simplicity we assume that the capacity of the battery is not a limiting factor.
- The processor must process a collection of n jobs of various sizes. Each job has an associated time interval, representing the interval between the time the job arrives in the system and a specified deadline.
- The objective is to determine the minimum recharge rate and schedule such that all jobs finish by their respective deadlines without depleting the battery.

We give a polynomial-time combinatorial algorithm for this problem of minimizing the recharge rate. Our algorithm can be viewed as a homotopic optimization algorithm that maintains an energy optimal schedule as the recharge rate continuously decreases. The first step is to again consider the natural linear program representation of this problem. Interpreting the complementary slackness conditions combinatorially yields four structural conditions that characterize and allow us to recognize an optimal schedule. The high level intuition behind our algorithm is to first start with an energy optimal schedule (computed

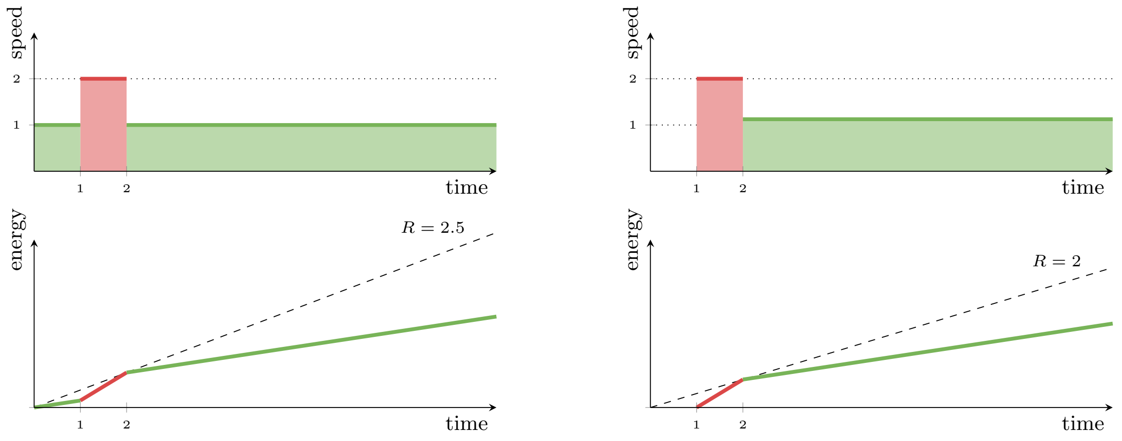


Figure 2: The energy optimal schedule and the recharge rate optimal schedule.

using [34]), and a recharge rate R such that this schedule satisfies the first three optimality conditions. The algorithm then lowers R, while maintaining a schedule satisfying the first three optimality conditions until the fourth condition is satisfied.

In order to build intuition on how we lower R continuously while maintaining certain structural properties, let us consider the following simple example instance as seen in Figure 9. The processor can be run at speed 1 with power 1, or at speed 2 with power 4. Job j_1 is released at time 0 with deadline 10 and work 9, and j_2 is released at time 1 with deadline 2 and work 2. The energy optimal schedule would run job j_2 at speed 2 during the time interval $[1, 2]$ and run job j_1 at speed 1 during the time intervals $[0, 1]$ and $[2, 10]$. The minimal recharge rate at which this schedule is feasible is $R = 2.5$. However one can achieve a smaller recharge rate by moving some of the processing done on j_1 during the time interval $[0, 1]$ to the time interval $[2, 10]$. This trend of moving processing of work later in time to allow the battery to recharge is the essence of how we are able to lower the recharge rate. To formalize this intuition we need to determine how to find such paths and rates as well as determine how long we can move work until we must recalculate.

One can discretize the continuous algorithm by calculating, given the transfer rates for the current transfer path collection, the next structural event and then discretely transferring enough work to reach that next event. To obtain convergence to the optimal solution and a polynomial bound on the runtime, we design our algorithm so as to maintain a hierarchy of monotonicity invariants. Combining this monotonicity with a polynomial bound on the number of structural states yields the desired polynomial bound.

2.0 FRACTIONAL FLOW AND ENERGY TRADE-OFF SCHEDULES

In this chapter we consider how to schedule jobs on such a speed-scalable processor in order to obtain an optimal trade-off between a natural performance measure (fractional weighted flow) and the energy used. Using the notation introduced in Chapter 1, we give a polynomial time algorithm for `FE-FDWA`. Before giving an outline for this Chapter let us review the setting. Fully formal definitions are given in Section 2.2. We need to explain how we model the processors, the jobs, a schedule, our performance measure, and the energy-performance trade-off:

The Speed-Scalable Processor: We assume that the processor can operate in any of a discrete set of modes, each with a specified speed and corresponding power consumption.

The Jobs: Each job has a release time when the job arrives in the system, a volume of work (think of a unit of work as being an infinitesimally small instruction to be executed), and a total importance or weight. The ratio of the weight to the volume of work specifies the density of the job, which is the importance per unit of work of that job.

A Schedule: A schedule specifies the job that is being processed and the mode of the processor at any point in time.

Our Performance Measure: The fractional weighted flow of a schedule is the total over all units of work (instructions) of how much time that work had to wait from its release time until its execution on the processor, times the weight (aggregate importance) of that unit of work. So work with higher weight is considered to be more important. Presumably the weights are specified by higher-level applications that have knowledge of the relative importance of various jobs.

Optimal Trade-off Schedule: An optimal trade-off schedule minimizes the fractional weighted flow plus the energy used by the processor (energy is just power integrated over time). To gain intuition, assume that at time zero a volume p of work of weight w is released. Intuitively/Heuristically one might think that the processor should operate in the mode i that minimizes $w\frac{p}{2s_i} + P_i\frac{p}{s_i}$, where s_i and P_i are the speed and power of mode i respectively, until all the work is completed; In this schedule the time to finish all the work is $\frac{p}{s_i}$, the fractional weighted flow is $w\frac{p}{2s_i}$, and the total energy usage is $P_i\frac{p}{s_i}$. So the larger the weight w, the faster the mode that the processor will operate in. Thus intuitively the application-provided weights inform the system scheduler as to which mode to operate in so as to obtain the best trade-off between energy and performance. (The true optimal trade-off schedule for the above instance is more complicated as the speed will decrease as the work is completed.)

In Section 2.1 we explain the relationship of our result to related results in the literature. Unfortunately both the design and analysis of our algorithm are complicated, so in Section 2.3 we give an overview of the main conceptual ideas before launching into details in the subsequent sections. In Section 2.4 we present the obvious linear programming formulation of the problem, and discuss our interpretation of information that can be gained about optimal schedules from both the primal and dual linear programs. In Section 2.6 we use this information to develop our algorithm. Finally in Section 2.7 we analyze the running time of our algorithm.

2.1 RELATED RESULTS

To the best of our knowledge there are three papers in the algorithmic literature that study computing optimal energy trade-off schedules. All of these papers assume that the processor can run at any non-negative real speed, and that the power used by the processor is some nice function of the speed. Essentially both [2, 31] give polynomial time algorithms for the special case of our problem where the densities of all units of work are the same. In [31], Pruhs et al. give a homotopic optimization algorithm that, intuitively, traces out all schedules that are Pareto-optimal with respect to energy and fractional flow, one of which must obviously be the optimal energy trade-off schedule. [2] give a dynamic programming algorithm and deserve credit for introducing the notion of trade-off schedules. [13] give a polynomial-time algorithm for recognizing an optimal schedule. They also showed that the optimal schedule evolves continuously as a function of the importance of energy, implying that a continuous homotopic algorithm is, at least in principle, possible. However, [13] was not able to provide any bound, even exponential, on the time of this algorithm, nor was [13] able to provide any way to discretize this algorithm.

To reemphasize, the prior literature [2, 13, 31] on our problem assumes that the set of allowable speeds is continuous. Our setting of discrete speeds both more closely models the current technology, and seems to be algorithmically more challenging. In [13] the recognition of an optimal trade-off schedule in the continuous setting is essentially a direct consequence of the KKT conditions of the natural convex program, as it is observed that there is essentially only one degree of freedom for each job in any plausibly optimal schedule, and this degree of freedom can be recovered from the candidate schedule by looking at the speed that the job is run at. In the discrete setting, we shall see that there is again essentially only one degree of freedom for each job. But, unfortunately, one cannot easily recover the value of this degree of freedom by examining the candidate schedule. Thus we do not know of any simple way to even recognize an optimal trade-off schedule in the discrete setting.

2.2 MODEL & PRELIMINARIES

We consider the problem of scheduling a set $\mathcal{J} := \{ 1, 2, \dots, n \}$ of n jobs on a single processor featuring k different speeds $0 < s_1 < s_2 < \dots < s_k$. The power consumption of the processor while running at speed s_i is $P_i \geq 0$. We use $\mathcal{S} := \{ s_1, s_2, \dots, s_k \}$ to denote the set of speeds and $\mathcal{P} := \{ P_1, P_2, \dots, P_k \}$ to denote the set of powers. While running at speed s_i, the processor performs s_i units of work per time unit and consumes energy at a rate of P_i.

Each job $j \in \mathcal{J}$ has a release time r_j, a processing volume (or work) p_j, and a weight w_j. Moreover, we denote the value $d_j := \frac{w_j}{p_j}$ as the density of job j. For each time t, a schedule S must decide which job to process and at what speed. Preemption is allowed, that is, a job may be suspended at any point in time and resumed later on. We model a schedule S by a speed function $V \colon \mathbb{R}_{\geq 0} \to \mathcal{S}$ and a scheduling policy $J \colon \mathbb{R}_{\geq 0} \to \mathcal{J}$. Here, $V(t)$ denotes the speed at time t, and $J(t)$ the job that is scheduled at time t. Jobs can be processed only after they have been released. For job j let $I_j = J^{-1}(j) \cap \big[r_j, \infty\big)$ be the set of times during which it is processed. A feasible schedule must finish the work of all jobs. That is, the inequality $\int_{I_j} S(t)\,\mathrm{d}t \geq p_j$ must hold for all jobs j.

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

We define $s_0 \coloneqq 0$, $P_0 \coloneqq 0$, $s_{k+1} \coloneqq s_k$, and $P_{k+1} \coloneqq \infty$ to simplify notation. Note that, without loss of generality, we can assume $\frac{P_i - P_{i-1}}{s_i - s_{i-1}} < \frac{P_{i+1} - P_i}{s_{i+1} - s_i}$; Otherwise, any schedule using s_i could be improved by linearly interpolating the speeds s_{i-1} and s_{i+1}. Most of the time, our analysis assumes all densities to be distinct. This is without loss of generality, and we explain at the end of our analysis section (see Section 2.6.3) how the algorithm can be changed to handle jobs of equal densities.

2.3 OVERVIEW

In this section we give an overview of our algorithm design and analysis. We first outline how we extract geometric information from the primal-dual formulation of the problem, and then give an example of this geometric information, providing insight into how this yields an optimality condition. Finally, we give a first overview of how to leverage this condition when designing our algorithm.

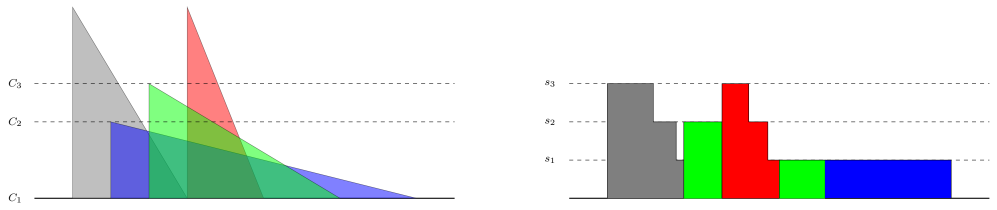


Figure 3: The dual lines for a 4-job instance, and the associated schedule.

We start by considering a natural linear programming formulation of the problem. We then consider the dual linear program. Using complementary slackness we find necessary and sufficient conditions for a candidate schedule to be optimal. Reminiscent of the approach used in the case of continuous speeds in [13], we then interpret these conditions in the following geometric manner. Each job j is associated with a linear function $D_j^{\alpha_j}(t)$, which we call a *dual line*. This dual line has a slope of $-d_j$ and passes through point (r_j, α_j), for some $\alpha_j > 0$. Here t is time, α_j is the dual variable associated with the primal constraint that all the work from job j must be completed, r_j is the release time of job j, and d_j is the density of job j. Given such an α_j for each job j, one can obtain an associated schedule as follows: At every time t, the job j being processed is the one whose dual line is the highest at that time, and the speed of the processor depends solely on the height of this dual line at that time.

The left picture in Figure 9 shows the dual lines for four different jobs on a processor with three modes. The horizontal axis is time. The two horizontal dashed lines labeled by C_2 and C_3 represent the heights where the speed will transition between the lowest speed mode and the middle speed mode, and the middle speed mode and the highest speed mode, respectively (these lines only depend on the speeds and powers of the modes and not on the jobs). The right picture in Figure 9 shows the associated schedule.

By complementary slackness, a schedule corresponding to a collection of α_j's is optimal if and only if it processes exactly p_j units of work for each job j. Thus we can reduce finding an optimal schedule to finding values for these dual variables satisfying this property.

Our algorithm is a primal-dual algorithm that raises the dual α_j variables in an organized way. We iteratively consider the jobs by decreasing density. In iteration i, we construct the optimal schedule S_i for the i most dense jobs from the optimal schedule S_{i-1} for the $i-1$ most dense jobs. We raise the new dual variable α_i from 0 until the associated schedule processes p_i units of work from job i. At some point raising the dual variable α_i may cause the dual line for i to "affect" the dual line for a previous job j in the sense that α_j must be raised as α_i is raised in order to maintain the invariant that the right amount of work is processed on job j. Intuitively one might think of "affection" as meaning that the dual lines intersect (this is not strictly correct, but it is a useful initial geometric interpretation to gain intuition). More generally this affection relation can be transitive in the sense that raising the dual variable α_j may in turn affect another job, etc.

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

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

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

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

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

2.4 STRUCTURAL PROPERTIES VIA PRIMAL-DUAL FORMULATION

This section derives the structural optimality condition (Theorem 4) on which our algorithm is based. We do so by means of an integer linear programming (ILP) description of our problem. Before we give the ILP and derive the condition, we show how to divide time into discrete time slots with certain properties.

Discretizing Time. We define time slots in which the processor runs at constant speed and processes at most one job. Note that, in general, these time slots may be arbitrarily small, yielding an ILP with many variables. At first glance, this seems problematic, as it renders a direct solution approach less attractive. However, we are actually not interested in solving the resulting ILP directly. Instead, we merely strive to use it and its dual in order to obtain some simple structural properties of an optimal schedule.

To this end, consider $\varepsilon > 0$ and let $T \in \mathbb{N}$ be such that $T\varepsilon$ is an upper bound on the completion time of non-trivial[1] schedules (e.g., $T\varepsilon \geq \max_j(r_j + \sum_j {}^{p_j}/_{s_1})$). Given a fixed problem instance, there is only a finite number of jobs and, without loss of generality, an optimal schedule performs only a finite number of speed switches and preemptions. Thus, we can choose $\varepsilon > 0$ such that

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

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

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

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

ILP & Dual Program. Let the indicator variable x_{jti} denote whether job j is processed in slot t at speed s_i. Note that T as defined above is an upper bound on the total number of time slots. This allows us to model our scheduling problem via the ILP given in Figure 4a. The first set of constraints ensures that all jobs are completed, while the second set of constraints ensures that the processor runs at constant speed and processes at most one job in each time slot.

In order to use properties of duality, we consider the relaxation of the above ILP. It can

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

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

(a) ILP formulation of our problem.

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

(b) Dual of the ILP's relaxation.

Figure 4: Primal-dual formulation of our problem.

easily be shown that an optimal schedule can use highest density first as its scheduling policy. Therefore, there is no advantage to scheduling partial jobs in any time slot. It follows that by considering small enough time slots, the value of an optimal solution to the LP will be no less than the value of the optimal solution to the ILP. After considering this relaxation and taking the dual, we get the dual program shown in Figure 4b.

The complementary slackness conditions of our primal-dual program are

$$\alpha_j > 0 \quad \Rightarrow \quad \sum_{t=r_j}^{T}\sum_{i=1}^{k} x_{jti}\cdot s_i = p_j, \tag{2.1}$$

$$\beta_t > 0 \quad \Rightarrow \quad \sum_{j\in\mathcal{J}}\sum_{i=1}^{k} x_{jti} = 1, \tag{2.2}$$

$$x_{jti} > 0 \quad \Rightarrow \quad \beta_t = \alpha_j s_i - P_i - s_i d_j(t - r_j + {}^{1}\!/\!_{2}). \tag{2.3}$$

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

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

Definition 1 (Dual Lines & Upper Envelope). *For a value $a \geq 0$ and a job j we define the linear function $D_j^a \colon [r_j, \infty) \to \mathbb{R}, t \mapsto a - d_j(t - r_j)$ as the* dual line *of j with offset a.*

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

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

Definition 2 (Left Upper Envelope & Discontinuity). *Given a job set $H \subseteq \mathcal{J}$ and upper envelope of H, UE_H, we define the* left upper envelope *at a point t as the limit of UE_H as we approach t from the left. That is, the left upper envelope of H is the function $\mathrm{LUE}_H \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, t \mapsto \lim_{t' \to t^-} \mathrm{UE}_H(t')$. Note that an equivalent definition of the left upper envelope is $\mathrm{LUE}_H(t) = \max_{j \in H : r_j < t}\bigl(D_j^{a_j}(t), 0\bigr)$.*

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

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

Definition 3 (Line Schedule). *Consider dual lines $D_j^{a_j}$ for all jobs. The corresponding* line schedule *schedules job j in all intervals $I \subseteq [r_j, \infty)$ of maximal length in which j's dual line is on the upper envelope of all jobs (i.e., $\forall t \in I \colon D_j^{a_j}(t) = \mathrm{UE}(t)$). The speed of a job j scheduled at time t is s_i, with i such that $C_i = \check{C}(D_j^{a_j}(t))$.*

See Figure 9 for an example of a line schedule. Together with the complementary slackness conditions, we can now easily characterize optimal line schedules.

Theorem 4. *Consider dual lines $D_j^{a_j}$ for all jobs. The corresponding line schedule is optimal* *with respect to fractional weighted flow plus energy if it schedules exactly p_j units of work for each job j.*

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

2.5 AFFECTION & RAISING PROCESS

Recall the algorithmic idea sketched in Section 2.3: We aim to develop a primal-dual algorithm that raises dual variables in a structured fashion. Theorem 4 provides us with some motivation for how to organize this raising. A first approach might be to raise the dual line of a new job i, leaving the dual lines of previously scheduled jobs untouched. At some point, its dual line will claim enough time on the upper envelope to be fully processed. However, in doing so we may affect (i.e., reduce) the time windows of other (already scheduled) jobs. Thus, while raising i's dual line, we must keep track of any affected jobs and ensure that they remain fully scheduled. This section formalizes this idea by defining affections and by structuring them in such a way that we can efficiently keep track of them.

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

2.5.1 Affected Jobs

Let us define a relation capturing the idea of jobs affecting each other while being raised.

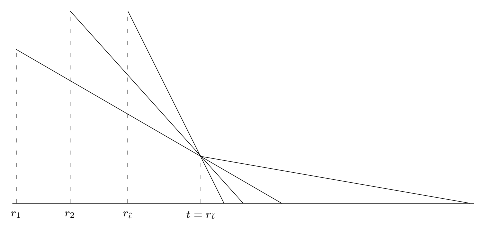
Definition 5 (Affection)**.** *Consider two different jobs j and j'. We say that job j* affects *job j' at time τ if raising (only) the dual line D_j^τ would decrease the processing time of j' in the corresponding line schedule.*

We write $j \to j'$ to indicate that j affects j' (and refer to the parameter τ separately, if not clear from the context). Similarly, we write $j \not\to j'$ to state that j does not affect j'. Before we show how to structure the affection between different jobs, let us collect some simple observations on when and how jobs can actually affect each other. We start with observing that jobs can affect each other only if their dual lines intersect on the upper envelope or left upper envelope (see Figure 5).

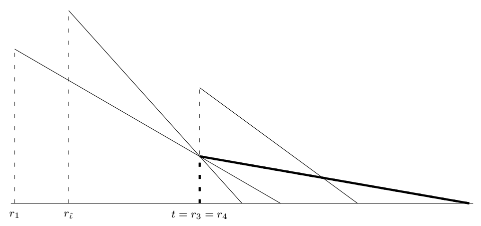
Observation 6. *Given jobs j and j' with $j \to j'$, their dual lines must intersect on the upper envelope, or on the left upper envelope at a discontinuity. That is, if t is the intersection point of j and j', we have either $D_j^\tau(t) = D_{j'}^\tau(t) = \mathrm{UE}^\tau(t)$, or $D_j^\tau(t) = D_{j'}^\tau(t) = \mathrm{LUE}^\tau(t)$ and t is a discontinuity. Further there must be some $\epsilon > 0$ such that j' is run in either $(t-\epsilon, t)$ or $(t, t+\epsilon)$.*

The following two observations are the counterpart of Observation 6. More precisely, Observation 7 states that only the most and least dense jobs intersecting at the upper envelope can be affected (Figure 5). Similarly, Observation 8 states that only the highest density job intersecting at the left upper envelope can be affected (Figure 5).

Observation 7. *For $t \in \mathbb{R}_{\geq 0}$ consider the maximal set H_t of jobs that intersect the upper envelope at t and define $H_t^- := H_t \cap \{\, j \mid r_j < t \,\}$. Let $\check{\imath} \in H_t$ denote the job of lowest density and let $\hat{\imath} \in H_t^-$ denote the job of highest density in the corresponding sets (assuming the sets are nonempty). The following hold:*



(a) Affections: $\{1,2,\hat\imath\} \to \{\check\imath\}$, $\{1,2\} \to \{\hat\imath\}$, and $\{1,2,\hat\imath,\check\imath\} \not\to \{1,2\}$.



(b) Affections: $1 \to \hat\imath$ and $\{1,\hat\imath,3\} \not\to \{1,3\}$. (Job 3 is denoted by the thick line.)

Figure 5: Illustration of Observation 7 and Observation 8.

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

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

(c) For all $j \in H_t$ and $j' \in H_t \setminus \{\check\imath, \hat\imath\}$ we have $j \not\to j'$.

Observation 8. *For $t \in \mathbb{R}_{\geq 0}$ consider the maximal set H_t of jobs that intersect the left upper envelope at t where t is a discontinuity. Define $H_t^- := H_t \cap \{j \mid r_j < t\}$. Let $\hat\imath \in H_t^-$ denote the job of highest density in H_t^- (assuming it is nonempty). The following hold:*

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

(b) For all $j \in H_t$ and $j' \in H_t \setminus \{\hat\imath\}$ we have $j \not\to j'$.

The final two observations are based on the fact that once the dual lines of a higher density job and a lower density job intersect, the higher density job is "dominated" by the lower density job (Figure 6).

Observation 9. *No job $j \in [i-1]$ can intersect a job j' of lower density at its own release time r_j.*

Observation 10. *Given jobs j and j' with $d_j > d_{j'}$ that intersect on the upper envelope or left upper envelope at point t, we have that $\mathrm{UE}^\tau(t') \geq \mathrm{LUE}^\tau(t') \geq D_{j'}^\tau(t') > D_j^\tau(t')$, for all $t' > t$.*

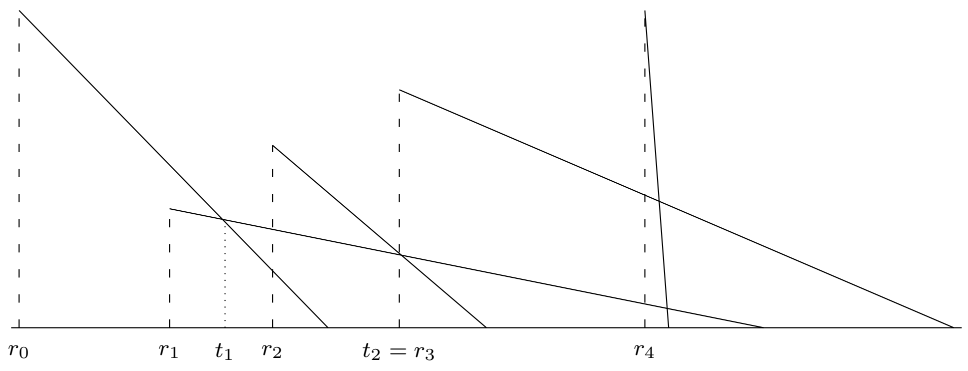


Figure 6: Observation 10. Both the upper envelope and the left upper envelope cases: note that $D_0^\tau(t') < \mathrm{LUE}^\tau(t') \leq \mathrm{UE}^\tau(t')$ for all $t' > t_1$, and $D_2^\tau(t') < \mathrm{LUE}^\tau(t') \leq \mathrm{UE}^\tau(t')$ for all $t' > t_2$.

2.5.2 Structuring the Affection

Equipped with these observations, we provide additional structural properties about how different jobs can affect each other. Assume jobs to be ordered by decreasing density and fix a job i. In the following, we study how the raising of job i can affect the already scheduled jobs in $\{ 1, 2, \ldots, i-1 \}$. We will use our insights later to prove that the graph induced by the affection relations forms a tree (see Lemma 17).

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

Lemma 11. *Consider two jobs $j_0 \in \mathcal{L}_l$ and $j_+ \in \mathcal{L}_{l+1}$ with $j_0 \to j_+$. Then job j_+ has a larger density than job j_0. That is, $d_{j_+} > d_{j_0}$.*

Proof. We prove the statement of the lemma by induction. The base case $l = 0$ is trivial, as i has the lowest density of all jobs and, by construction, $\mathcal{L}_0 = \{ i \}$. Now consider the case $l \geq 1$ and let $j_- \in \mathcal{L}_{l-1}$ be such that $j_- \to j_0$. By the induction hypothesis, we have $d_{j_0} > d_{j_-}$. For the sake of a contradiction, assume $d_{j_+} < d_{j_0}$. Let t_1 denote the intersection point of j_0 and j_-, and let t_2 denote the intersection point of j_0 and j_+. By Observation 6, these intersection points lie on the upper envelope or the left upper envelope. We consider three cases:

Case $t_1 > t_2$: Because of $t_1 > t_2$, the assumption $d_{j_0} > d_{j_+}$ implies $D^\tau_{j_0}(t_1) < D^\tau_{j_+}(t_1) \leq \mathrm{LUE}^\tau(t_1) \leq \mathrm{UE}^\tau(t_1)$ by Observation 10. This contradicts Observation 6.

Case $t_1 < t_2$: Because of $t_2 > t_1$, the induction hypothesis $d_{j_0} > d_{j_-}$ implies, by Observation 10, $D^\tau_{j_0}(t_2) < D^\tau_{j_-}(t_2) \leq \mathrm{LUE}^\tau(t_2) \leq \mathrm{UE}^\tau(t_2)$. This contradicts Observation 6.

Case $t_1 = t_2$: First note that this intersection point must lie on the upper envelope since otherwise it would lie on the left upper envelope at a discontinuity and, by Observation 8, $j_0 \not\to j_+$. Further, because $j_0 \neq i$ and $d_{j_0} > d_{j_+}$, Observation 9 implies $r_{j_0} < t_1$. Together with $d_{j_+} < d_{j_0}$ and $j_0 \to j_+$, this implies that j_+ has minimal density among all jobs intersecting the upper envelope in t_1 (by Observation 7). We get $j_- \to j_+$ and, thus, $j_+ \in \mathcal{L}_l$. This contradicts j_+ being a level $l+1$ node.

□

Lemma 12. *Given two level l jobs $j_1, j_2 \in \mathcal{L}_l$, we have $j_1 \not\to j_2$ and $j_2 \not\to j_1$.*

Proof. The statement is trivial for $l = 0$, as $\mathcal{L}_0 = \{i\}$. For $l \geq 1$ consider $j_1, j_2 \in \mathcal{L}_l$ and assume, for the sake of a contradiction, that $j_1 \to j_2$ (the case $j_2 \to j_1$ is symmetrical). Let $\iota_1, \iota_2 \in \mathcal{L}_{l-1}$ with $\iota_1 \to j_1$ and $\iota_2 \to j_2$. By Lemma 11 we have $d_{\iota_1} < d_{j_1}$ and $d_{\iota_2} < d_{j_2}$. Let t_0 denote the intersection point of j_1 and j_2, t_1 the intersection point of j_1 and ι_1, and t_2 the intersection point of j_2 and ι_2. Analogously to the proof of Lemma 11, one can see that $t_0 = \min\{t_1, t_2\}$ (as otherwise at least one of these intersection points would not lie on the (left) upper envelope). We distinguish the following cases:

Case $t_0 = t_1 < t_2$: First note that t_1 and t_0 cannot lie on the left upper envelope at a discontinuity, since by Observation 8 either $j_1 \not\to j_2$ or $\iota_1 \not\to j_1$. So, by Observation 6, t_0 and t_1 lie on the upper envelope. Job j_2 must have minimal density among all jobs intersecting the upper envelope at t_1, as otherwise its intersection point with ι_2 cannot lie on the upper envelope. But then, by Observation 7, we have $\iota_1 \to j_2$. Together with Lemma 11 this implies $d_{j_2} > d_{\iota_1}$, contradicting the minimality of j_2's density.

Case $t_0 = t_2 < t_1$: By Observation 6 j_1, j_2 and ι_2 either lie on the left upper envelope or the upper envelope. Assume they are on the left upper envelope. Note that since ι_2 and

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

Case $t_0 = t_1 = t_2$: The same argument as in the first case shows that these points do not lie on the left upper envelope at a discontinuity but must lie on the upper envelope. Without loss of generality, assume $d_{j_1} > d_{j_2}$. We get $r_{j_1} < t_1$ (Observation 9). With $d_{j_2} > d_{\iota_2}$ and Observation 7 this implies $\iota_2 \not\to j_2$, contradicting the definition of ι_2.

□

Lemma 13. *A level l job cannot be affected by more than one job of a lower level.*

Proof. The statement is trivial for $l \in \{0, 1\}$. Thus, consider a job $j \in \mathcal{L}_l$ for $l \geq 2$ and let j_1 and j_2 be two different lower level jobs with $j_1 \to j$ and $j_2 \to j$. By definition, both j_1 and j_2 must be level $l-1$ jobs. Also, similar to previous proofs, we can see that all three jobs must intersect at the same point t on the upper envelope or left upper envelope. Let us first assume they intersect at the upper envelope. Observation 7 implies that j has maximal density of all jobs intersecting the upper envelope at t (as otherwise j can be affected by neither j_1 nor j_2, both having a lower density). Consider the lowest density job $\check{\iota}$ intersecting the upper envelope at t. By Observation 7, at least one among j_1 and j_2 must affect $\check{\iota}$. Assume, without loss of generality, it is j_1. This implies that $\check{\iota}$ has level $l' \leq l$. Actually, we must have $l' < l$, because otherwise $d_{\check{\iota}} < d_{j_1}$ would contradict Lemma 11. Similarly, $l' = l - 1$ would contradict Lemma 12. Thus, we have $l' \leq l - 2$. But since we have $\check{\iota} \to j$, we get a contradiction to j being a level l node.

Now assume that all three jobs intersect at a discontinuity of the left upper envelope. Observation 8 tells us that j must be the job of highest density intersecting at t. Assume without loss of generality that $d_{j_1} < d_{j_2}$. Then, by Observation 10, j_2 is not on the (left) upper envelope to the right of t. However, since it is not the root ($l \geq 2$), there must be some job $\breve{\imath}$ of smaller density that intersects j_2 on the (left) upper envelope to the left of t (its parent). This contradicts that j_2 would be on the left upper envelope at t. □

Lemma 14. *Consider two nodes $j_1 \in \mathcal{L}_{l_1}$ and $j_2 \in \mathcal{L}_{l_2}$ with $l_2 - l_1 \geq 2$. Then, we must have $j_2 \not\rightarrow j_1$.*

Proof. For the sake of a contradiction, assume $j_2 \rightarrow j_1$ and let j denote a level $l_2 - 1$ node with $j \rightarrow j_2$. Similarly to the previous proofs, we can see that all three jobs j_1, j_2, and j must intersect the upper envelope or left upper envelope at a common intersection point t. In the first case, assume this is on the upper envelope. Since $d_{j_2} > d_j$ and $j \rightarrow j_2$, we get that $r_{j_2} < t$ and that j_2 has maximal density among all jobs intersecting the upper envelope at t and having a release time before t (Observation 9 and Observation 7). We get $j_1 \rightarrow j_2$, contradicting j_2 being of at least level $l_1 + 2$.

In the case when they intersect at a discontinuity of the left upper envelope, Observation 8 implies either $j_2 \not\rightarrow j_1$ or $j \not\rightarrow j_2$, a contradiction. □

Lemma 15. *Consider two nodes $j_1 \in \mathcal{L}_{l_1}$ and $j_2 \in \mathcal{L}_{l_2}$ with $l_2 = l_1 + 1$, and $j_1 \rightarrow j_2$. Then, if there exists a node $j_3 \in \mathcal{L}_{l_1}$ such that $j_2 \rightarrow j_3$, it must be that $j_3 = j_1$.*

Proof. For the sake of contradiction, assume there exists a node $j_3 \neq j_1$ such that $j_3 \in \mathcal{L}_{l_1}$ and $j_2 \rightarrow j_3$. First, note that by Lemma 11 we have $d_{j_1} < d_{j_2}$. Let t_1 be the intersection of j_1 and j_2, and t_2 be the intersection of j_2 and j_3. There are two cases to consider. In the first case, assume $t_1 = t_2$ and note that the intersection must lie on the upper envelope: if it were on the left upper envelope at a discontinuity, $j_1 \rightarrow j_2$ and $j_2 \rightarrow j_3$ would contradict Observation 8. Since $d_{j_1} < d_{j_2}$, either j_1 or j_3 is the job with lowest density. Then, since $t_1 = t_2$, by Observation 7 either $j_1 \rightarrow j_3$ or $j_3 \rightarrow j_1$. Both cases contradict Lemma 12 since $j_1, j_3 \in \mathcal{L}_{l_1}$.

In the second case, assume $t_1 \neq t_2$. If $t_1 < t_2$, then, since $d_{j_1} < d_{j_2}$ by Observation 10, $D^{\tau}_{j_2}(t') < \mathrm{LUE}^{\tau}(t') \leq \mathrm{UE}^{\tau}(t')$ for all $t' > t_1$ which contradicts j_2 being on the (left) upper envelope at t_2. For the case $t_1 > t_2$, a similar argument shows that j_2 must be the least dense job that intersects the upper envelope at t_2, as otherwise $D^{\tau}_{j_2}(t_1) < \mathrm{LUE}^{\tau}(t_1) \leq \mathrm{UE}^{\tau}(t_1)$. If the jobs meet on the upper envelope at t_2, Observation 7 yields $j_3 \rightarrow j_2$. Together with

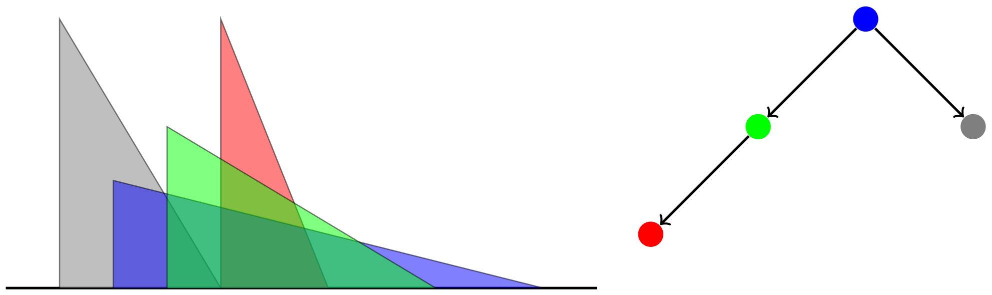


Figure 7: Affection tree example, rooted at the blue job.

$j_1 \to j_2$ and $j_3 \neq j_1$, this contradicts Lemma 13. If the jobs meet on the left upper envelope at t_2, we see similarly to previous proofs that j_3 is not the root, cannot be processed to the right of t_2 (since j_2 is less dense), and its less dense parent muss intersect it to the left of t_2. But then, j_2 cannot be on the left upper envelope at t_2, a contradiction. □

2.5.3 Affection Tree

We will now formally define and study the graph defined by the affection relation. Using the lemmas from Section 2.5.2, we will show that this graph is a tree (Lemma 17).

Definition 16 (Affection Tree)**.** *Let $G_i(\tau)$ be the directed graph induced by the affection relation on jobs $1, 2, \ldots, i$. The* affection tree *is an undirected graph $A_i(\tau) = (V_i(\tau), E_i(\tau))$ where $j \in V_i(\tau)$ if and only if j is reachable from i in $G_i(\tau)$, and for $j_1, j_2 \in V_i(\tau)$ we have $(j_1, j_2) \in E_i(\tau)$ if and only if $j_1 \to j_2$ or $j_2 \to j_1$.*

See Figure 7 for an illustration of this definition.

Lemma 11 to Lemma 15 imply that, if we omit edge directions, this subgraph indeed forms a tree rooted at i such that all children of a node j are of higher density. We state and prove this in the next lemma.

Lemma 17. *Let A_i be the (affection) graph of Definition 16. Then A_i is a tree, and if we root A_i at i, then for any parent and child pair $(\iota_j, j) \in G$ it holds that $d_{\iota_j} < d_j$.*

Proof. Assume, for the sake of contradiction, that there exists a cycle C in G. Let v be a node in C that belongs to the highest level set, say $\mathcal{L}_{l_1}$. Note that such a v is unique since otherwise there would be two nodes in the same level with at least one having an affection to the other contradicting Lemma 12. Let v_1, v_2 be the neighbors of v in C and $v_3 \in \mathcal{L}_{l_1-1}$ be the node such that $v_3 \to v$ (note that it may be $v_3 = v_1$ or $v_3 = v_2$). Note that by Lemma 14 we also have $v_1, v_2 \in \mathcal{L}_{l_1-1}$. By Lemma 13, either $v_1 \not\to v$ or $v_2 \not\to v$. Assume without loss of generality this is v_1. Since v_1 is a neighbor of v in C and $v_1 \not\to v$, we have $v \to v_1$. However, this contradicts Lemma 15. □

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

2.6 COMPUTING AN OPTIMAL SCHEDULE

In this section we describe and analyze the algorithm for computing an optimal schedule. We introduce the necessary notation and provide a formal definition of the algorithm in Section 2.6.1. In Section 2.6.2, we prove the correctness of the algorithm. Finally, Section 2.6.3 explains how the algorithm and the analysis need to be adapted to allow for arbitrary (not pairwise different) densities.

2.6.1 Preliminaries and Formal Algorithm Description

Our algorithm will use the affection tree to track the jobs affected by the raising of the current job i and compute corresponding raising rates. The raising will continue until job i is completely scheduled, or there is some structural change causing us to recompute the rates at which we are raising dual lines. For example a change in the structure of the affection tree when new nodes are affected will cause us to pause and recompute. The intuition for each event is comparatively simple (see Definition 18), but their formalization is quite technical, requiring us to explicitly label the start and ending points of each single execution interval of each job. To do so, we introduce the following *interval notation*. See Figure 8 for a corresponding illustration.

Interval Notation. Let $\hat{r}_1, \hat{r}_2, \ldots, \hat{r}_n$ denote the jobs' release times in non-decreasing order. We define Ψ_j as a set of indices with $q \in \Psi_j$ if and only if job j is run between $\hat{r}_q$ and $\hat{r}_{q+1}$ (or after $\hat{r}_n$ for $q = n$). Job j must run in a (sub-)interval of $\big[\hat{r}_q, \hat{r}_{q+1}\big)$. Let $x_{\ell,q,j}$ denote the left and $x_{r,q,j}$ denote the right border of this execution interval. Let $s_{\ell,q,j}$ denote the speed at which j is running at the left endpoint corresponding to q and $s_{r,q,j}$ denote the speed j is running at the right endpoint. Let $q_{\ell,j}$ be the smallest and $q_{r,j}$ be the largest indices of Ψ_j, i.e., the indices of the first and last execution intervals of j.

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

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

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

Events & Algorithm. Given this notation, we now define four different types of events which intuitively represent the situations in which we must change the rate at which we are raising the dual line. We assume that from τ until an event we raise each dual line at a constant rate. More formally, we fix τ and for $j \in [i]$ and $u \geq \tau$ let $\alpha_j(u) = \alpha_j(\tau) + (u - \tau)\alpha'_j(\tau)$.

Definition 18 (Event)**.** *For $\tau_0 > \tau$, we say that an* event occurs at τ_0 *if there exists $\epsilon > 0$ such that at least one of the following holds for all $u \in (\tau, \tau_0)$ and $v \in (\tau_0, \tau_0 + \epsilon)$:*

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

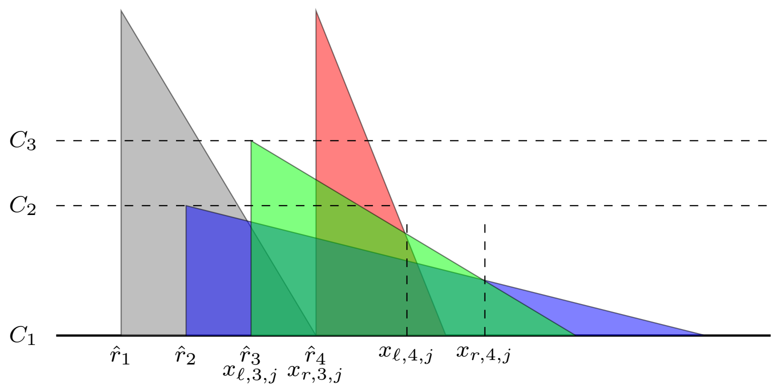


Figure 8: Let j be the green job. The set $\Psi_j := \{3, 4\}$ corresponds to the 2 execution intervals of j. The speeds at the border of the first execution interval, $s_{\ell,3,j}$ and $s_{r,3,j}$, are both equal to s_2. Similarly, the border speeds in the second execution interval, $s_{\ell,4,j}$ and $s_{r,4,j}$, are both equal to s_1. The value $q_{\ell,j} = 3$ refers to the first and $q_{r,j} = 4$ to the last execution interval of j. Finally, the indicator variables for j in the depicted example have the following values: $y_{r,j}(3) = y_{\ell,j}(3) = 1$ (borders at some release time), $y_{r,j}(4) = y_{\ell,j}(4) = 0$ (borders not at some release time), $\chi_j(3) = 1$ (not j's last execution interval), and $\chi_j(4) = 0$ (last execution interval of j).

and a $q \in \Psi_i(\tau)$ such that $x_{r,q,i}(u) < r_j$ and $x_{r,q,i}(v) = r_j$. This is called a simple rate change event*.*

- *Job i completes enough work, i.e., $p_i(u) < p_i < p_i(v)$. This is called a* job completion event*.*

A formal description of the algorithm can be found in Algorithm 4.1.

2.6.2 Correctness of the Algorithm

In this subsection we focus on proving the correctness of the algorithm. Throughout this subsection, we assume that the iteration and value of τ are fixed. Recall that we have to raise the dual lines such that the total work done for any job $j \in [i-1]$ is preserved. To calculate the work processed for j in an interval, we must take into account the different speeds at which j is run in that interval. Note that the intersection of j's dual line with the i-th speed

```
for each job $i$ from 1 to $n$:
    while $p_i(\tau) < p_i$:                         {job $i$ not yet fully processed in current schedule}
        for each job $j \in A_i(\tau)$:
            calculate $\delta_{j,i}(\tau)$                              {see Equation (2.5)}
        let $\Delta\tau$ be the smallest $\Delta\tau$ returned by any of the subroutines below:
            (a) JobCompletion($S(\tau), i, [\alpha'_1, \alpha'_2, \ldots, \alpha'_i]$)          {time to job completion}
            (b) AffectionChange($S(\tau), A_i(\tau), [\alpha'_1, \alpha'_2, \ldots, \alpha'_i]$)   {time to affection change}
            (c) SpeedChange($S(\tau), [\alpha'_1, \alpha'_2, \ldots, \alpha'_i]$)                {time to speed change}
            (d) RateChange($S(\tau), i, [\alpha'_1, \alpha'_2, \ldots, \alpha'_i]$)               {time to rate change}
        for each job $j \in A_i(\tau)$:
            raise $\alpha_j$ by $\Delta\tau \cdot \delta_{j,i}$
        set $\tau = \tau + \Delta\tau$
        update $A_i(\tau)$ if needed                  {only if Case (b) returns the smallest $\Delta\tau$}
```

Listing 2.1: The algorithm for computing an optimal schedule.

threshold C_i occurs at $t = \frac{\alpha_j - C_i}{d_j} + r_j$. Therefore, the work done by a job $j \in [i]$ is given by

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

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

$$p'_j = \sum_{q \in \Psi_j} \left[s_{\ell,q,j} \left(\frac{\alpha'_j}{d_j} - x'_{\ell,q,j} \right) + s_{r,q,j} \left(x'_{r,q,j} - \frac{\alpha'_j}{d_j} \right) \right]. \tag{2.4}$$

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

$$\delta_{j,\iota_j} := \Bigg(1 + y_{\ell,j}(q_{\ell,j}) \frac{d_j - d_{\iota_j}}{d_j} \frac{s_{\ell,q_{\ell,j},j} - s_{r,\rho_j(q_{\ell,j}),j}}{s_{r,q_{r,j},j}} \\ + \sum_{(j,j')\in A_i} \left((1-\delta_{j',j}) \frac{d_j - d_{\iota_j}}{d_{j'} - d_j} \frac{s_{\ell,q_{j,j'},j}}{s_{r,q_{r,j},j}} + \frac{d_j - d_{\iota_j}}{d_j} \frac{s_{\ell,q_{j,j'},j} - s_{r,\rho(q_{j,j'}),j}}{s_{r,q_{r,j},j}} \right) \Bigg)^{-1}. \tag{2.5}$$

We will prove in Lemma 21 that these rates are work-preserving for all jobs $j \in [i-1]$. Note that the algorithm actually uses $\delta_{j,i}$ which we can compute by taking the product of the $\delta_{k,k'}$ over all edges (k,k') on the path from j to i. Similarly we can compute $\delta_{j,j'}$ for all $j, j' \in A_i$.

Lemma 19. *Intersection points on the upper envelope cannot move towards the right when τ is increased.*

Proof. Since, by Lemma 17, parents in the affection tree are always of lower-density than their children, and since dual lines are monotonically decreasing, we have that $\delta_{\iota_j,j} \le 1$. This implies the claim. □

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

Lemma 20. *Consider any job $j \in A_i$ whose dual line gets raised at a rate $\delta_{j,i}$.*

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

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

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

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

Proof.

(a) Note that since $y_{\ell,j}(q) = 1$, this implies that $x_{\ell,q,r} = r_j$. Since by Lemma 19 intersection points can only move towards the left and by definition D^τ_j is defined in $[r_j, \infty)$ the statement follows.

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

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

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

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

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

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

(c) Note that since (j, j') is an edge in the affection tree, by Observation 6 we have that $D^{\tau}_{j'}$ and D^{τ}_{j} must intersect on the (left) upper envelope. Since $D^{\tau}_{j'}(t) = \alpha_{j'} - d_{j'}(t - r_{j'})$ and $D^{\tau}_{j}(t) = \alpha_j - d_j(t - r_j)$, the dual lines for j and j' intersect at

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

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

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

□

Equation 2.4 defines a system of differential equations. In the following, we first show how to compute a work-preserving solution for this system (in which $p'_j = 0$ for all $j \in [i-1]$) if $\alpha'_i = 1$, and then show that the corresponding τ values can be easily computed.

Lemma 21. *For a parent and child $(\iota_j, j) \in A_i$, set $\alpha'_j = \delta_{j,\iota_j}\alpha'_{\iota_j}$, and for $j' \notin A_i$ set $\alpha_{j'} = 0$. Then $p'_j = 0$ for $j \in [i-1]$.*

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

For a parent and child $(\iota_j, j) \in A_i$, we set $p'_j = 0$ in Equation 2.4 and solve for ${}^{\alpha'_j}/_{\alpha'_{\iota_j}} = \delta_{j,\iota_j}$. Let $I_{q,j} = \{\, q, \ldots, \rho_j(q) \,\}$ if $q \in \Psi_j$ and $\emptyset$ otherwise. We call $I_{q,j}$ a *maximal execution interval* of j if $I_{q-1,j} \cap I_{q,j} = \emptyset$. Let $M = \{\, q \in \Psi_j \mid I_{q,j} \text{ is max. execution interval of } j \,\}$. We have that $\bigcup_{q \in M} I_{q,j} = \Psi_j$. Let $p'_{j,S}$ where $S \subseteq \Psi_j$ be the rate of change of p_j due to the rate of change of the endpoints of the intervals in S. If j is run at its release time, then $y_{\ell,j}(q_{\ell,j}) = 1$ and by, Observation 7, j cannot intersect any of its children at its release time, so by Lemma 20

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

For any other $q \in M$ (including $q_{\ell,j}$ if $y_{\ell,j}(q_{\ell,j}) = 0$), q must begin at the intersection point of j and one of its children. That is, there exists a unique $(j, j') \in A_i$ such that $q = q_{j,j'}$. Therefore, by Lemma 20

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

For any $q \in M$, we have (Lemma 20) that $x'_{r,\rho(q),j} = 0$ if $\rho(q) \neq q_{r,j}$ and $x'_{r,q_{r,j},j} = -\frac{\alpha'_{\iota_j} - \alpha'_j}{d_j - d_{\iota_j}}$. The lemma follows by observing that $p'_j = \sum_{q \in M} p'_{j,I_{q,j}}$, the fact that j must intersect each of its children exactly once on the (left) upper envelope, and that for $(j, j') \in A_i$, we have that $\alpha_{j'}/\alpha_j = \delta_{j',j}$. □

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

Lemma 22. *An affection change event occurs at time τ_0 if and only if at least one of the following occurs.*

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

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

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

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

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

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

Let τ_0 be a time when an edge between j and j' is removed, and assume that j and j' had an intersection point t. Assume furthermore, without loss of generality, that j is a parent of j'. Therefore the affection $j \to j'$ ceases to exist at τ_0 (perhaps also $j' \to j$ if it existed). First note that at t, $D^{\tau_0}_j(t) = D^{\tau_0}_{j'}(t)$ must be on the upper envelope or left upper envelope at a discontinuity, otherwise there exists some $\epsilon > 0$ such that at time $\tau_0 - \epsilon$ their intersection was not on the upper envelope or the left upper envelope but the edge $j \to j'$ existed, contradicting Observation 6. We handle these two cases separately.

Subcase: $D^{\tau_0}_j(t) = D^{\tau_0}_{j'}(t)$ lies on upper envelope.

By Lemma 17, we know that it must be the case that $d_{j'} > d_j$. Furthermore, by Observation 7, since now it is the first time that $j \not\to j'$, either $r_j = t$ (which is covered in statement (a) of the lemma), or at least three jobs intersect at t. If this is the case, let j'' be the highest density job among the jobs that intersect at t. Note that j'' cannot be j (since $d_{j'} > d_j$) and it can also not be j' (since $j \not\to j'$).

By Observation 7, just before τ_0, j' does work to the left of the intersection point (between j and j') and j to the right. But at τ_0, j' cannot do any work directly to the left of t, because of j''. It follows that the interval of j' has disappeared, since to the left of t, j'' is run and to the right of t, j is run. This case is covered in the statement (b) of the lemma.

Subcase: $D_j^{\tau_0}(t) = D_{j'}^{\tau_0}(t)$ lies on left upper envelope at discontinuity t.

Note that since the intersection points do not move towards the right as τ increases (by Lemma 19), the intersection of j and j' was either at t or it was moving to the left towards t for all times during which $j \to j'$. However, since there is a discontinuity at t, there is some job j'' on the upper envelope that is not on the left upper envelope. If the intersection was at t then $j \to j'$ would not be possible. Therefore, there must exist some $\epsilon > 0$ such that at $\tau_0 - \epsilon$ the intersection of j and j' was below the curve of j''. This contradicts Observation 6. It follows that this subcase cannot occur.

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

Let τ_0 be the time when a new edge is added to the affection tree. First note that, without loss of generality, at least one new edge is between two nodes j and j' with (a) j is in the affection tree immediately before τ_0, (b) j' is not in the affection tree immediately before τ_0, and (c) j becomes the parent of j' at τ_0. Indeed, obviously at least one old node j of the affection tree must be involved as a parent in one of the new edges. Note that in the above cases "immediately before" may refer to either some interval $(\tau_0 - \epsilon, \tau_0)$, for an appropriate $\epsilon > 0$, or to the situation directly after some edge is removed at τ_0, and before adding the new edge.

If all new children j' of such old nodes were in the affection tree immediately before τ_0, there would also be some edge removal at τ_0 (as an additional edge would break the tree property, contradicting Lemma 17). This would reduce this case to the previous one, i.e., we consider edge removals at τ_0 before edge additions at τ_0.

From the above we get $d_j < d_{j'}$ (j being a parent of j'). Moreover, by Observation 6, at τ_0 the intersection point t of j and j' is on the (left) upper envelope and j' is run either to the left or right of t. Since j has a lower density, it must be run to the left of t. This is the case covered by statement (c) of the lemma.

□

2.6.2.1 The Subroutines There are four types of events that cause the algorithm to recalculate the rates at which it is raising the dual lines. In Lemma 22 we gave necessary and sufficient conditions for affection change events to occur. The conditions for the remaining event types to occur follow easily from Lemma 19 and Lemma 20. Given the rates at which the algorithm raises the dual lines, we can easily calculate the time until the next event. This subsection gives a formal description of these subroutines and their correctness proofs.

Job Completion Event. Job completion events, which capture when the current job i is finished, are the easiest events to handle. As long as no other event occurs, the work of job i is processed at a constant rate p_i', which can be computed by Equation (2.4) (using Lemma 20 to compute $x'_{l,q,i}$ and $x'_{r,q,i}$). With $p_i(\tau)$ denoting the work of i processed at the current time τ, we define

$$\Delta\tau \coloneqq \frac{p_i - p_i(\tau)}{\sum_{q\in\Psi_i(\tau)}\left[s_{\ell,q,i}\left(\frac{\alpha_i'(\tau)}{d_i} - x'_{\ell,q,i}(\tau)\right) + s_{r,q,i}\left(x'_{r,q,i}(\tau) - \frac{\alpha_i'(\tau)}{d_i}\right)\right]}. \tag{2.6}$$

With the above discussion, we immediately get the following lemma.

Lemma 23. *Assume the next event is a job completion event. Then this event occurs at $\tau + \Delta\tau$, with $\Delta\tau$ as computed by our job completion subroutine via Equation 2.6.*

Simple Rate Change Event. Simple rate change events are similarly easy to compute. These occur when the right side of i's last execution interval reaches the release time of some job. By Lemma 20(d), this happens only when the rate at which this interval border changes jumps from $1/d_i$ to 0. Thus, the corresponding time is computed as

$$\Delta\tau \coloneqq (\hat{r}_{q_{r,i}(\tau)+1}(\tau) - x_{r,q_{r,i}(\tau),i}(\tau)) \cdot d_i. \tag{2.7}$$

This yields the following lemma.

Lemma 24. *Assume the next event is a simple rate change event. Then this event occurs at $\tau + \Delta\tau$, with $\Delta\tau$ as computed by our simple rate change subroutine via Equation 2.7.*

Speed Change Event. While the basic idea is the same as for the previous events, computations for speed change events are a bit more tedious. For each execution interval of each job, we have to check when the job's dual function value at the interval borders crosses a speed threshold. See Algorithm 2.2 for the actual computations. We prove its correctness in Lemma 25.

Lemma 25. *Assume the next event is a speed change event. Then this event occurs at $\tau + \Delta\tau$, with $\Delta\tau$ as computed by our speed change subroutine shown in Algorithm 2.2.*

Proof. Consider the innermost "for loop" of Algorithm 2.2. It computes the time until the left or right border of the q-th execution interval of j reaches a speed threshold, assuming that no other event occurs before. To see this, note that if the interval has length 0 and not increasing in size (the *else* branch), no work will be done in this interval until the next event. Thus, in this case q cannot cause the next event. Otherwise (the *if* branch), we know that $x'_{\ell,q,j}(\tau) \leq 0$ and $\alpha'_j(\tau) \geq 0$, and these rates remain constant until the next event. Thus, the dual function value $D_j^{\tau'}(x_{\ell,q,j}(\tau'))$ at the left border of q is always non-decreasing for $\tau' > \tau$ (until the next event). Thus, the speed $s_{\ell,q,j}$ at this interval border remains constant until $\tau' > \tau$ with $D_j^{\tau'}(x_{\ell,q,j}(\tau')) = \hat{C}(D_j^{\tau}(x_{\ell,q,j}(\tau)))$. With $\Delta\tau_{\ell,q,j} = \tau' - \tau$, we can write

$$\begin{aligned}
D_j^{\tau'}(x_{\ell,q,j}(\tau')) &= \alpha_j(\tau') - d_j(x_{\ell,q,j}(\tau') - r_j) \\
&= \alpha_j(\tau) + \Delta\tau_{\ell,q,j}\alpha'_j(\tau) - d_j\bigl(x_{\ell,q,j}(\tau) + \Delta\tau_{\ell,q,j}x'_{\ell,q,j}(\tau) - r_j\bigr) \\
&= D_j^{\tau}(x_{\ell,q,j}(\tau)) + \Delta\tau_{\ell,q,j}\bigl(\alpha'_j(\tau) - d_j x'_{\ell,q,j}(\tau)\bigr).
\end{aligned}$$

If we set this equal to $\hat{C}(D_j^{\tau}(x_{\ell,q,j}(\tau)))$ and solve for $\Delta\tau_{\ell,q,j}$, we get exactly the value computed by Algorithm 2.2. Analogously, we see that $\Delta\tau_{r,q,j}$ are computed correctly. Finally, the algorithm set $\Delta\tau$ to the first of all these computed events. □

Affection Change Event. The last event type is the most involved. However, Lemma 22 gives the exact conditions for when and why an affection change event can occur. More precisely, Lemma 22(a) and 22(b) correspond to edge removals in the affection tree, while Lemma 22(c) corresponds to an addition of an edge. The computations of all such possible events is formalized in Algorithm 2.3. Lemma 26 states and proves its correctness.

```
for each job $j \in [i]$:
    for each $q \in \Psi_j(\tau)$:
        if $x_{r,q,j} \neq x_{\ell,q,j}$ or $x'_{r,q,j} - x'_{\ell,q,j} > 0$:
            $\Delta\tau_{\ell,q,j} = \frac{\hat{C}(D^\tau_j(x_{\ell,q,j}(\tau))) - D^\tau_j(x_{\ell,q,j}(\tau))}{-x'_{\ell,q,j}(\tau)d_j + \alpha'_j(\tau)}$
            $\Delta\tau_{r,q,j} = \frac{\hat{C}(D^\tau_j(x_{r,q,j}(\tau))) - D^\tau_j(x_{r,q,j}(\tau))}{-x'_{r,q,j}(\tau)d_j + \alpha'_j(\tau)}$
        else:
            $\Delta\tau_{\ell,q,j} = \Delta\tau_{r,q,j} = \infty$
$\Delta\tau = \min_{j\in[i], q\in\Psi_j(\tau)}(\min(\Delta\tau_{\ell,q,j}, \Delta\tau_{r,q,j}))$
return $\Delta\tau$
```

Listing 2.2: SpeedChange($S(\tau), [\alpha'_1, \alpha'_2, \ldots, \alpha'_i]$).

Lemma 26. *Assume the next event is an affection change event. Then this event occurs at $\tau + \Delta\tau$, with $\Delta\tau$ as computed by our affection change subroutine shown in Algorithm 2.3.*

Proof. By Lemma 22, an edge (and hence a job) is removed from the affection tree only when a nonzero interval becomes 0. Thus for any job j in the tree and nonzero interval q of it, the rate of change of the size of that interval is $v = x'_{r,q,j}(\tau) - x'_{\ell,q,j}(\tau)$, which is negative if the size of the interval is decreasing. The size of the interval is $x_{r,q,j}(\tau) - x_{\ell,q,j}(\tau)$, thus at rate v it will become zero at $\Delta\tau_{j,q}$.

By Lemma 22, an edge (and hence job) is added to the affection tree only when a job j not in the affection tree intersects a job a in the affection tree at the right endpoint of a nonzero interval of j. Since j is not in the affection tree, its endpoints do not change as τ increases. For some interval q, the distance between D^τ_a and D^τ_j at the right endpoint of q is $D^\tau_j(x_{r,q,j}(\tau)) - D^\tau_a(x_{r,q,j}(\tau))$, and D^τ_a increases at a rate of α'_a. □

2.6.2.2 Completing the Correctness Proof We are now ready to prove the correctness of the algorithm. We handle termination in Theorem 28, where we prove a polynomial running time for our algorithm.

Theorem 27. *Assuming that Algorithm 4.1 terminates, it computes an optimal schedule.*

```
{ $all\ \tau_{j,q}\ and\ \tau_{j,q,a}\ initialized\ with\ \infty$ }
for $j \in A_i(\tau)$:                                   { calculate affection tree removals }
    for $q \in \Psi_j(\tau)$:
        if $x_{\ell,q,j}(\tau) \neq x_{r,q,j}(\tau)$ and $x'_{r,q,j}(\tau) - x'_{\ell,q,j}(\tau) < 0$:      { q shrinks }
            $\Delta\tau_{j,q} = \frac{x_{r,q,j}(\tau) - x_{\ell,q,j}(\tau)}{x'_{\ell,q,j}(\tau) - x'_{r,q,j}(\tau)}$
$\Delta\tau_1 = \min_{j \in A_i(\tau), q \in \Psi_j(\tau)}(\Delta\tau_{j,q})$

for $j \in [i] \setminus A_i(\tau)$:                     { calculate affection tree additions }
    for $q \in \Psi_j(\tau)$:
        for $a \in A_i(\tau)$:
            if $x_{\ell,q,j} \neq x_{r,q,j}$ and $r_a < x_{r,q,j}$:
                $\Delta\tau_{j,q,a} = \frac{D_j^\tau(x_{r,q,j}(\tau)) - D_a^\tau(x_{r,q,j}(\tau))}{\alpha'_a}$

$\Delta\tau_2 = \min_{j \in [i] \setminus A_i(\tau), q \in \Psi_j(\tau), a \in A_i(\tau)}(\Delta\tau_{j,q,a})$
$\Delta\tau = \min(\Delta\tau_1, \Delta\tau_2)$
return $\Delta\tau$
```

Listing 2.3: AffectionChange$(S(\tau), A_i(\tau), [\alpha'_1, \alpha'_2, \ldots, \alpha'_i])$.

Proof. The algorithm outputs a line schedule S, so by Theorem 4, S is optimal if for all jobs j the schedule does exactly p_j work on j. We now show that this is indeed the case.

For a fixed iteration i, we argue that a change in the rate at which work is increasing for j (i.e., a change in p'_j) may occur only when an event occurs. This follows from Equation 2.4, since the rate only changes when there is a change in the rate at which the endpoints of intervals move, when there is a change in the speed levels employed in each interval, or when there is an affection change (and hence a change in the intervals of a job or a change in α'_j). These are exactly the events we have defined. It can be shown that the algorithm recalculates the rates at any event (proofs deferred to the full version), and by Lemma 21 it calculates the correct rates such that $p'_j(\tau) = 0$ for $j \in [i-1]$ and for every τ until some τ_0 such that $p_i(\tau_0) = p_i$, which the algorithm calculates correctly (proof also deferred to the full version). Thus we get the invariant that after iteration i we have a line schedule for the first i jobs that does p_j work for every job $j \in [i]$. The theorem follows. □

2.6.3 A Note on Density Uniqueness

For two jobs i and j with different densities $d_i \neq d_j$, the dual lines $D_i^{a_i}$ and $D_j^{a_j}$ intersect in at most one point t^*. Therefore the only time they can both be on the upper envelope (or left upper envelope) is t^*. However, if $d_i = d_j$ and $D_i^{a_i}$ and $D_j^{a_j}$ intersect once, then they intersect at every time after both i and j have been released, and thus both may be on the upper envelope for entire intervals. We resolve any ambiguity by imposing the rule that if $d_i = d_j$, $r_i < r_j$, and $D_i^{a_i}$ and $D_j^{a_j}$ intersect, then i must complete all its work before j completes any work (if $r_i = r_j$, we arbitrarily pick one to complete first). Let $J = \{ j_1, \ldots, j_z \}$ be the largest set of jobs that all have some density d_j and intersect, ordered by release time. We say that the intersection between $D_{j_i}^{a_{j_i}}$ and $D_{j_{i+1}}^{a_{j_{i+1}}}$ occurs at the time at which j_i has completed p_{j_i} work, if such a time exists. For any other pair of jobs $j_i, j_\ell \in J$ such that $|i - \ell| > 1$, we say that $D_{j_i}^{a_{j_i}}$ and $D_{j_\ell}^{a_{j_\ell}}$ do not intersect. Thus for the purpose of having a well-defined upper envelope, we consider a job $j_\ell \in J$ to be on the upper envelope at t only when $D_{j_\ell}^{a_\ell}$ is the highest curve at t, every other job $j_i \in J$ with $i \in [\ell - 1]$ has completed by t, and either j_ℓ has not completed by t or $\ell = z$.

To consider how the algorithm must be modified, fix an iteration i of the algorithm and a τ. If $(j, \iota_j) \in A_i$ with $d_j = d_{\iota_j}$, we set $\delta_{j,\iota_j} = 1$. All that remains is to show how the rate of change of the intersection point between j and ι_j (as defined above) can be computed, i.e., $x'_{\ell,q_{j,\iota_j},j}$, since now Lemma 20(c) calculates a value that is not well-defined. This is calculated such that $p'_j = 0$. In other words, by Equation 2.4 and Lemma 20(b) we have that

$$x'_{\ell,q_{j,\iota_j},j} = -\frac{1}{s_{r,q_{r,j},j}} \left(\sum_{q \in \Psi_j} (s_{\ell,q,j} - s_{r,q,j}) \frac{\alpha'_j}{d_j} - s_{\ell,q,j} x'_{\ell,q,j} \right). \tag{2.8}$$

Thus if we know the rates of change of the intersection points for j and its children, we can calculate the rate of change of j's intersection with ι_j. If $(\iota_j, u) \in A_i$ and $d_{\iota_j} > d_u$, then $\delta_{\iota_j,u}$ will be calculated differently than shown in Equation 2.5, but the necessary change is a simple replacement of the term $(1 - \delta_{j,\iota_j})/(d_j - d_{\iota_j})$ with the term $x'_{\ell,q_{j,\iota_j},j}/\alpha'_{\iota_j}$.

2.7 THE RUNNING TIME

The purpose of this section is analyze the running time of Algorithm 4.1 by proving the following theorem.

Theorem 28. *Algorithm 4.1 takes $O(n^4 k)$ time.*

We use the following approach in order to prove Theorem 28. Details follow below.

- We give upper bounds on the total number of events that can occur in Lemma 30. This is relatively straightforward for job completion, simple rate change, and speed change events, which can occur $O(n)$, $O(n^2)$, and $O(n^2 k)$ times, respectively. However, bounding the number of times an affection change event can occur is more involved: One can show that whenever an edge is removed from the affection tree, there exists an edge which will never again be in the affection tree. This implies that the total number of affection change events is upper bounded by $O(n^2)$ as well.
- We show in Lemma 31 that the next event can always be calculated in $O(n^2)$ time.
- We show in Lemma 32 that the affection tree can be updated in $O(n)$ time after each affection change event.

The above results imply that our algorithm has a running time of $O(n^4 k)$, and therefore Theorem 28 follows.

We start with an auxiliary lemma that will be useful for bounding the number of affection change events in Lemma 30.

Lemma 29. *Consider some time τ_0 where an edge (j, j') is removed from the affection tree. Then, there exists some edge (u, v) that is also being removed at τ_0 such that (u, v) will not be present for all remaining iterations of the algorithm.*

Proof. First note that by the definition of the affection tree, it must be that the affection $j \to j'$ is being removed. Since j is a parent of j', by Lemma 17 we have $d_j < d_{j'}$. Also, by Lemma 22, this edge can be removed because either the intersection between j and j' becomes equal to r_j or two intersection points become equal. We handle these cases separately.

In the first case we show that the affection edge $j \to j'$ cannot be present again. To do this, we show that the invariant of j' not being processed on the (left) upper envelope to the right of r_j is always maintained. This implies that the edge $j \to j'$ is never present again. It is clearly true for τ_0 (say, in iteration i). Assume that for some iteration $i' \geq i$ this invariant is true. If j' is not being raised the invariant will remain true since curves can only be raised and not lowered. If j' is being raised, since it is not the lowest density job it must intersect some lower density job $j^{''}$ (its parent) that is also being raised. Further, since the invariant is true to this point, the intersection is not to the right of r_j. However, while j' is being raised, by Lemma 19 the intersection between j' and $j^{''}$ moves only to the left. Since $d_{j''} < d_{j'}$, j' will not be on the upper envelope or left upper envelope to the right of this intersection and the result follows.

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

Lemma 30. *The total number of events throughout the execution of the algorithm is $O\big(n^2 k\big)$.*

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

Job completion event: Note that a job completion event occurs exactly once per iteration: After a job completion event occurs on iteration i, we have that $p_i(\tau) = p_i$ and in turn the algorithm moves to the next iteration. Therefore, the total number of job completion events that occur is exactly n.

Simple rate change event: Consider iteration i. A simple rate change event can occur, if and only if, the last interval of job i changes from ending before the release time of some job to ending at the release time of the job. Note that since dual lines are never lowered during the execution of the algorithm this can occur at most once for each release time and therefore at most n times. As we have n iterations the total number of simple rate change events is $O(n^2)$

Affection change event: These events happen when an edge is either added or removed in the tree. Note that the number of edge additions is bounded by 2 times the number of edge removals, so it suffices to bound the number of removals. By Lemma 29, for each (possibly temporarily) removed edge at least one edge is removed permanently. Thus, the total number of such events is $O(n^2)$.

Speed change event: A speed change event occurs when the right or left endpoint of an interval for a job j crosses a speed threshold. We first show that each speed threshold can be crossed at most twice per interval, once by the left and once by the right endpoint of the interval. Consider an interval I for job j. If I is never removed, then each endpoint of I can only cross each speed threshold at most once, since by Lemma 19 the endpoints of intervals only move towards left, and furthermore no dual line is ever lowered during the execution of the algorithm. Else, if I is removed then the left and right endpoints of I coincide on the upper envelope at some point t (by Lemma 22) and there must be some other job j' of lower-density whose dual-line also intersects at t. However, by Lemma 19, the left endpoint of j' (in case more than two dual-lines intersect at t let j' be the job corresponding to such a dual line of lowest-density) will only move to the left while this interval is not present. Therefore, even if interval I does reappear, the left and right endpoints will not be at lower speeds.

Finally, since each job has at most n intervals and each such interval can cause at most $2k$ speed change events, the total number of speed change events is $O(n^2k)$

As we have a constant number of event types each occurring $O(n^2k)$ times the Lemma follows. □

Lemma 31. *Calculating the next event takes $O(n^2)$ time.*

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

To calculate the next event, we look at each event type and calculate how far in the future the next event of this type will occur. Then we just choose the event of the type that will happen sooner. Therefore it suffices to give bounds on the time required to calculate the next event of each type (cf. subroutines in Section 2.6.2.1).

Affection change event: An affection change event has to be either a removal or an addition (see Lemma 26). If it is a removal, then by the observation on the number of intervals, it can be calculated in $O(n)$ time. On the other hand if the next affection change event is an addition, then again by the above observation on the number of intervals $O\big(n^2\big)$-time is required.

Speed change event: For any fixed interval the next speed change event can be calculated in constant time. Therefore, by the observation on the number of intervals, we have that the next such event over all jobs can be computed in time $O(n)$,

Simple rate change event: $O(n)$-time is sufficient in order to identify $q_{r,i}$ and $\hat{r}_{q_{r,i}+1}$, and therefore also to calculate this type of event as well.

Job completion event: We have to calculate $y_{r,j}, y_{\ell,j}$ for each of the $O(n)$ intervals, identify i' and calculate $\delta_{i,i'}$. Therefore we can calculate the next job completion event in time $O(n)$.

Combining the above, we can calculate the next event in $O\big(n^2\big)$ time. □

Lemma 32. *Updating the affection tree takes $O(n)$ time.*

Proof. A simple way to update the affection tree is by recomputing it from scratch at each update. By Lemma 11, jobs in the tree always have a higher density than their parents. Further, by Observations 7 and 8, if a job j is on the upper envelope (or left upper envelope) at some time t and has release time before t, and $j' \neq j$ is the highest-density job on the upper envelope (left upper envelope) at time t, then $j \to j'$, and for any other job $j'' \neq j'$ of higher density than j on the upper envelope (left upper envelope), $j \not\to j''$. Therefore, for any job j, its children in the affection tree are those highest-density jobs that intersect it on the left endpoint of any of its intervals that begin after j's release. Thus, to compute the affection tree, we can iterate through each interval I of job i that begins afters its release, add as i's children the highest density jobs that intersect it at I's left endpoint, and recursively do the same for i's children. By the observation that there are at most $2n$ intervals, this takes at most $O(n)$ time. $\square$

3.0 COMPLEXITY RESULTS FOR FLOW BASED OPTIMIZATION

In this Chapter, we provide several results to fill in Table 2. In Section 3.2, we show `B-IDWU` and `B-IDUA` are NP-hard. In Section 3.3 we give several polynomial time algorithms. Finally, in Section 3.4, we give reductions between budget and flow plus β energy problems.

3.1 MODEL AND NOTATION

We consider n jobs $J = \{ 1, 2, \dots, n \}$ to be processed on a single, speed-scalable processor. In the *continuous* setting, the processor's energy consumption is modeled by a power function $P \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ mapping a speed s to a power $P(s)$. We require P to be continuous, convex, and non-decreasing. Other than that, we merely assume P to be "nice" in the sense that we can solve basic equations involving the power function and, especially, its derivative and inverse. In the *discrete* setting, the processor features only k distinct speeds $0 < s_1 < s_2 < \dots < s_k$, where a speed s_i consumes energy at the rate $P_i \geq 0$. Even in the discrete case, we will often use $P(s)$ to refer to the power consumption when "running at a speed $s \in (s_i, s_{i+1})$" in between the discrete speeds. This is to be understood as interpolating the speed $s = s_i + \gamma(s_{i+1} - s_i)$ (running for a γ fraction at speed s_{i+1} and a $1 - \gamma$ fraction at speed s_i), yielding an equivalent discrete schedule. Each job $j \in J$ has a release time r_j, a processing volume p_j, and a weight w_j. For each time t, a schedule S must decide which job to process at what speed. Preemption is allowed, so that a job may be suspended and resumed later on. We model a schedule S by a speed function $\mathcal{V} \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ and a scheduling policy $\mathcal{J} \colon \mathbb{R}_{\geq 0} \to J$. Here, $\mathcal{V}(t)$ denotes the speed at time t, and $\mathcal{J}(t)$ the job that is scheduled at time t. Jobs can be processed only after they have been released. For job j let $I_j = \mathcal{J}^{-1}(j) \cap [r_j, \infty)$ be the set of times during which it is processed. A feasible schedule must finish the work of all jobs. That is, the inequality $\int_{I_j} \mathcal{V}(t)dt \geq p_j$ must hold for all jobs j.

We measure the quality of a given schedule S by means of its energy consumption and its fractional or integral flow. The energy consumption of a job j is $E_j = \int_{I_j} P(\mathcal{V}(t))dt$, and the energy consumption of schedule S is $\sum_{j \in J} E_j$. The *integral flow* $F_j = w_j(C_j - r_j)$ of a job j is the weighted difference between its completion time C_j and release time r_j. The integral flow of schedule S is $F(S) = \sum_{j \in J} F_j$. In contrast, the *fractional flow* can be seen as the flow on a per workload basis (instead of per job). More formally, if $p_j(t)$ denotes the work of job j that is processed at time t, the fractional flow time of job j is $w_j \int_{r_j}^{\infty} (t - r_j) \frac{p_j(t)}{p_j} dt$. Our goal is to find energy-efficient schedules that provide a good (low) flow. We consider two different ways to combine these conflicting goals. In the *budget setting*, we fix an *energy budget* $B \geq 0$ and seek the minimal (fractional or integral) flow achievable with this energy. In the *flow plus energy setting*, we want to minimize a linear combination $F(S) + \beta E(S)$ of energy and (fractional or integral) flow.

3.2 HARDNESS RESULTS

This section proves NP-hardness for the problems `B-IDUA` and `B-IDWU`. The reductions are from the subset sum problem, where we are given n elements $a_1 \geq a_2 \geq \cdots \geq a_n$ with $a_i \in \mathbb{N}$ as well as a target value $A \in \mathbb{N}$ with $a_1 < A < \sum_{i=1}^{n} a_i$. The goal is to decide whether there is a subset $L \subseteq [n]$ such that $\sum_{i \in L} a_i = A$.

For both reductions, we define for each element a_i a job set J_i such that jobs of two different sets will not influence each other. Each J_i contains one low density job and one/several high density jobs. Starting from a *base schedule*, we choose the parameters such that investing roughly a_i energy into J_i improves its flow by roughly a_i. More precisely, when J_i gets a_i, additional energy can be used to decreases the flow at a rate $\gg 1/2$ per energy unit. Given substantially more or substantially less energy, additional energy decreases the flow at a rate of only $1/2$. We achieve this by ensuring that at about a_i energy, the schedule switches from finishing the low density after the high density jobs to finishing it before them. For an energy budget of A, we can define a target flow that is reached if and only if there is an $L \subseteq [n]$ such that $\sum_{i \in L} a_i = A$ (corresponding to job sets that are given about a_i extra energy).

We assume a processor with two speeds $s_1 = 1$ and $s_2 = 2$ and power consumption rates $P_1 = 1$ and $P_2 = 4$. For an isolated job of weight w, this means that increasing a workload of x from speed s_1 to s_2 increases the energy by x and decreases the flow by $w \cdot \frac{x}{2}$. For ease of exposition, we assume that the first job of each job group J_i is released at time 0. To ensure that jobs of different job sets do not influence each other, one can increase all release times of the job set J_i by the total workload of all previous job sets.

3.2.1 Hardness of B-IDUA

For $i \in [i]$, we define a job set $J_i = \{\, (i,1), (i,2) \,\}$ of two unit weight jobs and set $\delta = \frac{1}{a_1 n^2}$. The release time of job $(i,1)$ is $r_{i1} = 0$ and its size is $p_{i1} = a_i$. The release time of job $(i,2)$ is $r_{i2} = \frac{a_i}{2}$ and its size is $p_{i2} = 2\delta a_i$.

Definition 33 (Base Schedule)**.** *The* base schedule BS_i *schedules job* $(i,1)$ *at speed 1 and job* $(i,2)$ *at speed* 2. *It finishes job* $(i,1)$ *after* $(i,2)$*, has energy consumption* $E(\mathrm{BS}_i) = a_i + 4\delta a_i$*, and flow* $F(\mathrm{BS}_i) = a_i + 2\delta a_i$*.*

Note that BS_i is optimal for the energy budget $E(\mathrm{BS}_i)$. Consider an optimal schedule S for the jobs $J = \bigcup_{i=1}^{n} J_i$ (release times shifted such that they do not interfere) for the energy budget $B = \sum_{i=1}^{n} E(\mathrm{BS}_i) + A$. Let $L \subseteq [n]$ be such that $i \in L$ if and only if J_i gets at least $E(\mathrm{BS}_i) + a_i - 4\delta a_i = 2a_i$ energy in S. We can now proof the desired hardness result.

Theorem 34. *`B-IDUA` is NP-hard.*

Proof. We show that S has flow at most $F = \sum_{i=1}^{n} F(\mathrm{BS}_i) - (\frac{1}{2} + \delta)A$ if and only if $\sum_{i \in L} a_i = A$. For the first direction, given that $\sum_{i \in L} a_i = A$, note that the schedule that gives each job set J_i with $i \in L$ exactly $E(\mathrm{BS}_i) + a_i$ energy and each J_i with $i \notin L$ exactly $E(\mathrm{BS}_i)$ energy adheres to the energy budget and has flow exactly F. For the other direction, consider $i \in [n]$, let E_i be the total energy used to schedule J_i in S, and let $\Delta_i = E_i - E(\mathrm{BS}_i)$ the additional energy used with respect to the base schedule. Then, for $i \notin L$, the flow of J_i is

$F(\mathrm{BS}_i) - \frac{1}{2}\Delta_i$, yielding an average flow gain per energy unit of $1/2$. For $i \in L$, the flow gain per energy unit is 1 for the interval $[2a_i, 2a_i + 2\delta a_i)$ and $1/2$ otherwise. Thus, the maximum average flow gain is achieved for $E_i = 2a_i + 2\delta a_i$, where the energy usage is $E(\mathrm{BS}_i) + a_i - 2\delta a_i$ and the flow is $F(\mathrm{BS}_i) - a_i/2$. This yields a maximum average flow gain per energy unit of $\frac{a_i/2}{a_i - 2\delta a_i} = \frac{1}{2-4\delta}$. Using these observations, we now show that, if $\sum_{i\in L} a_i \neq A$, the schedule has either too much flow or uses too much energy. Let us distinguish two cases:

Case 1: $\sum_{i\in L} a_i < A$: Using $a_i, A \in \mathbb{N}$ and our observations, the flow decreases by at most (wrt. $\sum_{i=1}^{n} \mathrm{BS}_i$)

$$\frac{1}{2-4\delta}\sum_{i\in L} a_i + \frac{1}{2}\left(A - \sum_{i\in L} a_i\right) = \frac{1}{2}A + \frac{\delta}{1-2\delta}\sum_{i\in L} a_i \leq \frac{1}{2}A + \frac{\delta}{1-2\delta}(A-1) < \left(\frac{1}{2} + \delta\right)A.$$

The last inequality follows from $\delta = \frac{1}{a_1 n^2} < \frac{1}{2A}$.

Case 2: $\sum_{i\in L} a_i > A$: This implies $\sum_{i\in L} a_i \geq A + 1$. Note that even if all jobs $(i,2)$ with $i \in \{1, 2, \ldots, n\}$ are run at speed 1 instead of speed 2, the total energy saved with respect to the base schedules is at most $\sum_{i=1}^{n} 2\delta a_i \leq \frac{2}{n}$. By this and the previous observations, the additional energy used by S with respect to the base schedules is at least $(1-4\delta)\sum_{i\in L} a_i - \frac{2}{n} \geq \sum_{i\in L} a_i - \frac{6}{n} \geq A + 1 - \frac{6}{n} > A$. □

3.2.2 Hardness of B-IDWU

In the following, we assume[1] $a_i \leq 2a_{i'}$ for any $i, i' \in [n]$. For $i \in [n]$, we define a job set $J_i = \{(i,0), (i,1), \ldots, (i,n)\}$ of $n+1$ unit size jobs. The release time of job $(i,0)$ is $r_{i0} = 0$ and its weight is $w_{i0} = \frac{a_i}{n}$. For $j \in [n]$, the release time of job (i,j) is $r_{ij} = r_i = 1 - \frac{a_i}{2a_1^2}$ and its weight is $w_{ij} = w_i = 2na_1^3$.

Definition 35 (Base Schedule)**.** *The base schedule* BS_i *schedules job* $(i,0)$ *at speed 1 and all remaining jobs at speed 2. It finishes job* $(i,0)$ *after all other jobs. The energy consumption of* BS_i *is* $E(\mathrm{BS}_i) = 1 + 2n$ *and its flow is* $F(\mathrm{BS}_i) = w_{i0} \cdot \left(1 + \frac{n}{2}\right) + w_i \cdot \frac{n(n+1)}{4}$. *Note that this is the optimal flow for an energy budget of* $E(\mathrm{BS}_i)$.

[1]This can be seen by first reducing subset sum to a variant of the problem where you must take exactly n elements by adding new elements with 0 value, and then reducing this new problem to the desired variant by adding some large enough value to each element, and n times that value to the target value.

Let us gather some simple observations on the structure of optimal schedules:

Observation 36. *Consider an optimal schedule S for the job set J_i and a fixed energy budget B. Then, without loss of generality, we have:*

(a) If S does not finish $(i, 0)$ last, it finishes $(i, 0)$ not later than r_i.

(b) If S finishes $(i, 0)$ last and $B \geq E(\mathrm{BS}_i)$, all jobs (i, j) for $j \in [a_j]$ are run at speed 2.

Thus, providing the base schedule with additional energy x will reduce the flow by $x\frac{w_{i0}}{2}$, until the optimal schedule switches from finishing $(i, 0)$ last to finishing it first. It is easy to check that this switch happens when the additional energy is $x_i = \frac{a_i}{a_1^2} - \delta_i$ with $\delta_i = \frac{nw_{i0}}{w_i - w_{i0}}$. We formalize this change and how it affects the flow gain in the following Observation.

Observation 37. *Consider an optimal schedule for a fixed energy budget B. Then,*

(a) If $B \in \big[E(\mathrm{BS}_i), E(\mathrm{BS}_i) + x_i\big)$, increasing the budget by $\varepsilon > 0$ decreases the flow by $\varepsilon \cdot \frac{w_{i0}}{2}$.

(b) If $B \in \big[E(\mathrm{BS}_i) + x_i, E(\mathrm{BS}_i) + \frac{a_i}{a_1^2}\big)$, increasing the budget by $\varepsilon > 0$ decreases the flow by $\varepsilon \cdot \frac{w_i}{2}$.

(c) If $B \in \big[E(\mathrm{BS}_i) + \frac{a_i}{a_1^2}, E(\mathrm{BS}_i) + 1\big)$, increasing the budget by $\varepsilon > 0$ decreases the flow by $\varepsilon \cdot \frac{w_{i0}}{2}$.

Note that $E(\mathrm{BS}_i) + 1$ is the maximum energy we can invest into J_i (all jobs run at speed 2).

Consider an optimal schedule S for the jobs $J = \bigcup_{i=1}^{n} J_i$ (release times shifted such that they do not interfere) that has an energy budget of $B = \sum_{i=1}^{n} E(\mathrm{BS}_i) + \frac{A}{a_1^2}$ and adheres to the properties of Observation 36. Let $L \subseteq [n]$ be such that $i \in L$ if and only if J_i is given at least $E(\mathrm{BS}_i) + x_i$ energy in S. We are now ready to show the main part of the reduction:

Theorem 38. *`B-IDWU` is NP-hard.*

Proof. The schedule S with energy budget B has flow at most $F = \sum_{i=1}^{n} F(\mathrm{BS}_i) - \frac{A}{2} - \frac{1}{8n}$ if and only if $\sum_{i \in L} a_i = A$.

For the first direction, given that $\sum_{i \in L} a_i = A$, note that the schedule that gives each job set J_i with $i \in L$ exactly $E(\mathrm{BS}_i) + \frac{a_i}{a_1^2}$ energy and each J_i with $i \notin L$ exactly $E(\mathrm{BS}_i)$ energy adheres to the energy budget, and, using that $a_1 \leq 2a_i$ for all i, has flow

$$\sum_{i=1}^{n} F(\mathrm{BS}_i) - \sum_{i\in L} \frac{w_{i0}}{2}\left(\frac{a_i}{a_1^2} + n\right) = \sum_{i=1}^{n} F(\mathrm{BS}_i) - \frac{A}{2} - \sum_{i\in L} \frac{a_i w_{i0}}{2a_1^2}$$
$$\leq \sum_{i=1}^{n} F(\mathrm{BS}_i) - \frac{A}{2} - \sum_{i\in L} \frac{w_{i0}}{4a_1} \leq \sum_{i=1}^{n} F(\mathrm{BS}_i) - \frac{A}{2} - \frac{1}{8n} = F.$$

For the other direction, first observe that $B \geq \sum_{i=1}^{n} E(\mathrm{BS}_i)$. Thus, without loss of generality, each J_i will receive at least $E(\mathrm{BS}_i)$ energy (otherwise, taking excess energy from a $J_{i'}$ with energy $> E(\mathrm{BS}_{i'})$ yields at least an equally good schedule). We distinguish two cases:

Case 1: $\sum_{i\in L} a_i < A$

This implies that $\sum_{i\in L} a_i \leq A - 1$. We first observe that, w.l.o.g., any J_i with $i \in L$ gets at least $\frac{a_i}{a_1^2}$ energy. This follows from $B \geq \sum_{i=1}^{n} E(\mathrm{BS}_i) + \sum_{i\in L} \frac{a_i}{a_1^2}$ and Observation 37, as otherwise we can take energy from a $J_{i'}$ whose energy lies in the intervals $\big(E(\mathrm{BS}_i), E(\mathrm{BS}_i) + x_i\big)$ or $\big(E(\mathrm{BS}_i) + \frac{a_i}{a_1^2}, E(\mathrm{BS}_i) + 1\big)$ without decreasing the flow. Let $\Delta_i = E(J_i) - \frac{a_i}{a_1^2} \geq 0$ be the extra energy used on J_i. The first $\frac{a_i}{a_1^2}$ energy units used after the base schedule improve the flow of J_i by $\frac{w_{i0}}{2}\left(\frac{a_i}{a_1^2} + n\right)$. After that, each additional energy unit gains flow at a rate of at most $\frac{w_{i0}}{2}$ per energy unit (Observation 37(c)). Similarly, the energy used for J_i with $i \notin L$ gain flow at a rate of at most $\frac{w_{i0}}{2}$ per energy unit (Observation 37(a)). With $w_{i0} \leq w_{10}$ for all $i \in [n]$, we get that the flow improvement with respect to the base schedules is at most

$$\sum_{i\in L} \frac{w_{i0}}{2}\left(\frac{a_i}{a_1^2} + n\right) + \sum_{i\in L} \Delta_i \frac{w_{i0}}{2} + \left(\frac{A}{a_1^2} - \sum_{i\in L}\left(\frac{a_i}{a_1^2} + \Delta_i\right)\right)\frac{w_{10}}{2} \leq \sum_{i\in L} \frac{w_{i0} n}{2} + \frac{w_{10}}{2} \cdot \frac{A}{a_1^2}$$
$$\leq \frac{A}{2} - \frac{1}{2} + \frac{w_{10}}{2} \cdot \frac{A}{a_1^2} = \frac{A}{2} - \frac{1}{2} + \frac{A}{2na_1} \leq \frac{A}{2} - \frac{1}{2} + \frac{na_1}{2na_1} = \frac{A}{2}.$$

That is, the target flow is not achieved.

Case 2: $\sum_{i\in L} a_i > A$

This implies $\sum_{i\in L} \frac{a_i}{a_1^2} \geq \frac{A+1}{a_1^2}$. Using that $\delta_i = \frac{nw_{i0}}{w_i - w_{i0}} < \frac{a_i}{w_i/2} \leq \frac{1}{na_1^2}$, we get that the additional energy used with respect to the base schedules is at least

$$\sum_{i\in L}\left(\frac{a_i}{a_1^2} - \delta_i\right) \geq \frac{A+1}{a_1^2} - \sum_{i\in L} \delta_i > \frac{A+1}{a_1^2} - n\frac{1}{na_1^2} = \frac{A}{a_1^2}.$$

This is a contradiction to S adhering to the energy budget B. $\square$

3.3 POLYNOMIAL TIME ALGORITHMS

In this section we provide polynomial time algorithms for `FE-IDUU`, `FE-ICUU`, and `FE-FCWA`. The algorithm for `FE-ICUU` generalizes and makes slight modifications to the algorithm in [2] to handle arbitrary power functions. We also provide a new, simple, combinatorial algorithm for `FE-IDUU`. While by the results of Section 3.4.1 we could use the algorithm for `FE-ICUU` to solve `FE-IDUU`, the algorithm we provide has the advantages of not having the numerical qualifications of the algorithm for `FE-ICUU`, as well as providing some additional insight into the open problem `FE-IDUA`. The algorithm for `FE-FCWA` generalizes and makes slight modifications to the algorithm in Chapter 2 to handle arbitrary power functions.

3.3.1 An Algorithm for FE-IDUU

Here we give a polynomial time algorithm for `FE-IDUU`. We describe the algorithm for two speeds; it is straightforward to generalize it to k speeds. The algorithm relies heavily upon the fact that, when jobs are of unit size, the optimal completion ordering is always FIFO (since any optimal schedule uses the SRPT scheduling policy).[2] Before describing the algorithm, we provide the necessary optimality conditions in Lemma 41. They are based on the following definitions, capturing how jobs may affect each other.

Definition 39 (Lower Affection). *For a fixed schedule, a job j_1* lower affects *a job j_2 if there is some $\varepsilon > 0$ such that decreasing the speed of j_1 by any value in $(0, \varepsilon]$ increases the flow of j_2.*

Definition 40 (Upper Affection). *For a fixed schedule, a job j_1* upper affects *a job j_2 if there is some $\varepsilon > 0$ such that increasing the speed of j_1 by any value in $(0, \varepsilon]$ decreases the flow of j_2.*

Lemma 41. *Consider an optimal schedule S and two consecutive speeds s_1 and s_2. Define $\alpha = \frac{P_2 - P_1}{s_2 - s_1}$ and $\kappa = -(P_1 - \alpha s_1) \geq 0$. For any job j with (interpolated) speed $s_j \in [s_1, s_2]$ in S, we have (a) $s_j > s_1 \implies j$ lower affects at least κ jobs, and (b) $s_j < s_2 \implies j$ upper*

[2]In fact, a slightly more general result yields an optimal `FE-IDUA` schedule given access to an optimal completion ordering.

affects at most $\kappa - 1$ jobs.

Proof. We start with (a). For the sake of a contradiction, assume $s_j > s_1$ but j lower affects less than κ jobs. Thus, for any $\varepsilon > 0$, increasing j's completion time by ε increases the flow of at most $\kappa - 1$ jobs by ε. If the resulting schedule is S'. For $t = \frac{1}{s_j}$, the energy from S to S' decreases by

$$tP\left(1/t\right) - (t+\varepsilon)P\left(1/(t+\varepsilon)\right)t\left(\alpha/t + P_1 - \alpha s_1\right) - (t+\varepsilon)\left(\alpha/(t+\varepsilon) + P_1 - \alpha s_1\right)$$
$$= \alpha + tP_1 - t\alpha s_1 - \alpha - (t+\varepsilon)P_1 + (t+\varepsilon)\alpha s_1 = -\varepsilon(P_1 - \alpha s_1) = \kappa\varepsilon.$$

Therefore, the total change in the objective function is at most $(\kappa-1)\varepsilon - \kappa\varepsilon < 0$, contradicting the optimality of S. Statement *(b)* follows similarly by decreasing the completion time of j by ε. □

Observation 42. *Consider two arbitrary jobs j and j' in an arbitrary schedule S.*

(a) If j upper affects $j' \neq j$ and j does not run at s_2, j' must run at s_1.

(b) While you raise the speed of j, the number of its lower and upper affections can only decrease.

(c) If j upper affects j', then changing the speed of j' will not change j's affection on j'.

(d) Assume j runs at speed s_j and upper affects m jobs. Then, in any schedule where j's speed is increased (and all other jobs remain unchanged), j lower affects at most m jobs.

Our algorithm GreedyAffection initializes each job with speed s_1. Consider jobs in order of release times and let j denote the current job. While j upper affects at least κ jobs and is not running at s_2, increase its speed. When this condition no longer holds, update j to the next job (or terminate if $j = n$).

Theorem 43. GreedyAffection *solves **FE-IDUU** in polynomial time.*

Proof. Assume A is not optimal and let O be an optimal schedule that agrees with A for the most consecutive job speeds (in order of release times). Let j be the first job that runs at a different speed and let s_A and s_O be the job's speeds in A and O. We consider two cases: If $s_A > s_O$, Observation 42(a) implies that every job that is upper affected by j in O other than j itself is run at s_1. Consider the time during the execution of A when the speed of

j was at s_O. Since A continued to raise j's speed, j upper affected at least κ jobs at this point. Let J be this set of jobs. By Observation 42(c), j still upper affects all jobs $j' \in J$ in O. This contradicts the optimality of O (Lemma 41). For the second case, assume $s_A < s_O$. By Lemma 41, j upper affects less than κ jobs in A. When A stops raising j's speed, all jobs to the right run at s_1. Observations 42(b) and (d) imply that j lower affects less than κ jobs in O, contradicting O's optimality (Lemma 41). $\square$

3.3.2 An Algorithm for FE-ICUU

In this subsection we show that `FE-ICUU` is in P. Essentially, it is possible to modify the algorithm from [2] to work with arbitrary power functions. The main alteration needed is that, for certain power functions that would yield differential equations too complicated for the algorithm to solve, we use binary search to find solutions to a these equations rather than solve the equations analytically. The only additional restriction on the power function is that $P(s)/s$ is convex[3].

Theorem 44. *There is a polynomial time algorithm for solving `FE-ICUU`.*

We do not alter the dynamic program. Given n jobs ordered by release times $r_1 \leq r_2 \leq \cdots \leq r_n$, an optimal schedule S can still be decomposed into non-idling subschedules $S_1, S_2, \ldots, S_k$ such that:

(a) S_i (exclusively) covers a time interval I_i and schedules all jobs with $r_j \in I_i$,

(b) S_i idles only when all its jobs have been completed.

Assume for the moment that, given the corresponding job sets, we can efficiently compute such non-idling subschedules. Then, for two jobs j_1, j_2 with $r_{j_1} \leq r_{j_2}$, the dynamic program computes an optimal schedule for all jobs j with $r_j \in [r_{j_1}, r_{j_2+1})$ by either combining two smaller non-idling subschedules or by computing the non-idling subschedule for the Interval $[r_{j_1}, r_{j_2+1})$ (see [2, Section 5]).

To extend this approach to arbitrary power functions, we have merely to show how to

[3]We require this restriction for it to be clear that the KKT conditions are sufficient for optimality in this case. It is possible, however, that the KKT conditions are sufficient for optimality even without this restriction. We also note that it is possible to develop a polynomial-time algorithm without the extra restriction on P based on the lemmas from Section 3.4.1.

compute such non-idling subschedules in this more general case. In the case when P is simple enough (which we describe precisely below), the following lemma essentially replaces Lemma 5.1 and Lemma 5.2 of [2].

Lemma 45. *Consider n jobs with $0 = r_1 \leq r_2 \leq \cdots \leq r_n$ that can be scheduled in time T such that an optimal schedule does not idle before all jobs are completed. Then the speeds used in the optimal schedule can be computed as follows:*
(a) For each job j solve the following equation for s_j:

$$P'(s_j) \cdot s_j - P(s_j) - n + i - 1 - \lambda = 0. \tag{3.1}$$

(b) By substituting s_j as computed above, compute λ as the unique solution to

$$\sum_{j=1}^{n} \frac{1}{s_j} - T = 0. \tag{3.2}$$

Proof. Write down convex program with job speeds as variables. Solving the above equations yields primal and dual solutions that adhere to KKT conditions. □

Here, P must be differentiable at all points and such that Equation (3.1) is analytically solvable for s_j. If either of these conditions on P do not hold, it is possible to obtain a solution to the system specified by Equations (3.1) and (3.2) by binary searching over λ, and finding, for each job j, via binary search, s_j satisfying Equation (3.1), and checking if, for the obtained values of s_j, Equation (3.2) holds.

3.3.3 An Algorithm for FE-FCWA

This subsection shows that `FE-FCWA` is in P. Both in Chapter 2 and in [13] we notice it is possible to express optimal schedules as a graph of lines. Intuitively, the key step is to note that "hypopower lines" in [13] can be treated in a way similar to "dual lines" in Chapter 2, and that we can use the algorithm in Chapter 2 with the main change being that we binary search for events rather than solve for them explicitly. As is noted in [13], if we know the configuration of the optimal schedule (i.e., the order in which jobs run, as well as if each job completes before the release of the next job run), it is possible to write down a set of equations (which we call *work equations*) for the initial hypopower of each job (i.e., the y-intercept of the hypopower lines). The ability to solve the work equations combined with knowledge of the configuration of the optimal schedule then translates into the ability to find the optimal schedule.

The algorithm in Chapter 2 adds jobs to the schedule one at a time and finds the configuration of the optimal schedule for the current set of jobs by slowly modifying the schedule and keeping track of when the work equations change. The points at which new work equations must be derived are essentially points at which the configuration changes, and we call these points *events*. The algorithm in Chapter 2 calculates the next event by explicitly solving differential equations derived from the work equations. For an arbitrary, continuous power function, solving such differential equations analytically seems to be, in general, quite challenging. Instead, we use binary search to find when the work equations change. The result is that our modification of the algorithm in Chapter 2 is able to to find the configuration of the optimal schedule, assuming it is possible to solve the work equations.

Theorem 46. *There is a polynomial time algorithm for solving* `FE-FCWA`*.*

Proof. The algorithm is as follows: Given an optimal hypopower schedule for the κ highest-dense jobs in the instance, add the $\kappa + 1$st highest dense job j with initial hypopower $\tau = 0$. Repeat the following until we obtain a schedule such that j completes p_j work: Binary seach over values of τ to find the smallest τ at which some event occurs, or j does p_j total work. For each new value of τ, solve the work equations, checking if an event has occurred. At such a τ that an event occurs, compute a new set of work equations.

A significant part of the analysis in Chapter 2 is to bound by a polynomial the number of times an event occurs. We now briefly describe why essentially the same analysis holds in our case. The key technical point is Lemma 19 of Chapter 2, which says that, in any iteration as τ increases, the intersection point of any two hypopower lines can only move left in the hypopower graph. Although this observation is derived from the problem-specific differential equations in Chapter 2, it is clear that the same observation can be derived from the assumptions that P is increasing and convex, and that we consider the jobs in order of decreasing density With this in hand, the same technical lemmas from Chapter 2 can be proved in our case to obtain that the described algorithm runs in polynomial time. □

3.4 EQUIVALENCE REDUCTIONS

Here we provide the reductions to obtain the hardness and algorithmic results that are not proven explicitly. First, we reduce B-ICUU to FE-ICUU. Combined with the algorithm from Section 3.3.2, this shows that B-ICUU is in P. The second reduction is from any problem in the discrete power setting to the corresponding continuous variant. As a result, the hardness proofs from Section 3.2 for B-IDWU and B-IDUA imply that B-ICWU and B-ICUA are NP-hard. Our final reduction is from B-FCWA to FE-FCWA. As a result of the algorithm in Section 3.3.3, this shows that B-FCWA is in P.

3.4.1 Reduction from B-ICUU to FE-ICUU

We show that, given an algorithm for the flow plus energy variant, we can solve the energy budget variant of ICUU. The basic idea is to modify the coefficient β in the flow plus energy objective until we find a schedule that fully utilizes the energy budget B. This schedule gives the minimum flow for B. The major technical hurdles to overcome are that the power function P may be non-differentiable, and may lead to multiple optimal flow plus energy schedules, each using different energies. Thus, we may not find a corresponding schedule for the given budget, even if there is one. To overcome this, we define the *affectance* ν_j of a job j. Intuitively, ν_j represents how many jobs' flow will be affected by a speed change of j. We show that a job's affectance is, in contrast to its energy and speed, unique across optimal schedules and changes continuously in β. This will imply that job speeds change continuously in β (i.e., for small enough changes, there are two optimal schedules with speeds arbitrarily close). We also give a continuous transformation process between any two optimal schedules. This eventually allows us to apply binary search to find the correct β.

The remainder of this subsection is as follows. We start with some auxiliary lemmas which shed light on how exactly affectance and job speeds are correlated. Lemma 51 shows that a job's affectance does not decrease as its speed increases and vice versa. In Lemma 52, we show that speed is continuous in affectance (all speeds resulting from a small affectance change are close to some speed for the old affectance). In Observation 53, we note that if an affectance maps to multiple speeds, it in fact maps to a continuous interval of speeds. And Observation 54 captures that speeds mapping to a fixed affectance are continous in β (for a small enough change in β, all speeds mapping to the new affectance are arbitrarily close to one of the old speeds for that affectance). It is worth mentioning that these results hold even in the more general setting of arbitrary weights (in contrast to the other results). Lemma 55 is another auxiliary Lemma, showing that the affectance of each job is non-increasing in β. Given all these information about the correlation of affectance, job speeds, and the value β, we eventually finish the proofs of the major results stated in Section 3.4.1, namely Lemmas 56, 57, and 58 as well as Lemmas 60 and 61. Also here, it is worth mentioning that Lemma 56 holds even in the more general setting of arbitrary sizes and weights.

Definitions & Notation: We start with some formal definitions for this section and a small overview of what they will be used for in the remainder. Definition 49 (affectance) will be most central to this section, as it will be shown in Lemma 56 and Corollary 59 to characterize optimal schedules. It uses the *subdifferential*[4] $\partial P(s)$ to handle non-differentiable power functions P.

Definition 47 (Total Weight of Lower and Upper Affection)**.** *In any schedule, l_j and u_j are the total weight of jobs lower and upper affected by j, respectively (see Definitions 39 and 40).*

Definition 48 (Job Group)**.** *A* job group *is a maximal subset of jobs such that each job in the subset completes after the release of the next job. Let J_i denote the job group with the i-th earliest release time and W_i the total weight of J_i ($J_i = \emptyset$ and $W_i = 0$ if J_i does not exist). Job groups J_i and J_{i+1} are* consecutive *if the last job in J_i ends at the release time of the first job in J_{i+1}. We set the indicator $\zeta_i = 1$ if and only if J_{i+1} exists and J_i and J_{i+1} are consecutive.*

Definition 49 (Affectance Property)**.** *The ith job group of a schedule satisfies the* affectance

[4]Subdifferentials generalize the derivative of convex functions. $\partial P(s)$ is the set of slopes of lines through $(s, P(s))$ that lower bound P. It is closed, convex on the interior of P's domain, and non-decreasing if P is increasing [22].

property *if either $\zeta_{i+1} = 0$ or the $i+1$st job group also satisfies the affectance property, and there exists $\mathcal{N}^i$ such that for all $v^i \in \mathcal{N}^i$ and $j \in J_i$*

$$v^i \in [0, \zeta_{i+1}(\nu^{i+1} + W_{i+1})], \tag{3.3}$$

$$v_j = v^i + u_j, \text{ and} \tag{3.4}$$

$$v_j = s_j d - P(s_j) \text{ for some } d \in \partial P(s). \tag{3.5}$$

Here, $\nu^i = \max \mathcal{N}^i$ if job group i exists, and $\nu^i = 0$ otherwise. A schedule satisfies the affectance property if all job groups in the schedule satisfy the affectance property.

Definition 50 (Affectance of a Job). *The set of speeds satisfying Equation (3.5) for $v_j = \nu$ is $\mathcal{S}(\nu)$. For each job j in job group i satisfying the affectance property, the* affectance of job *j is $\nu_j = \nu^i + u_j$.*

Auxiliary Results: Correlation between Affectance & Job Speeds: We now present several auxiliary lemmas which shed light on how affectance and job speeds are correlated.

Lemma 51. *For a given problem instance, let j and k be jobs from any (possibly different) schedules satisfying the affectance property. The following properties hold:*
(a) $s_j < s_k \implies \nu_j \leq \nu_k$, and
(b) $\nu_j < \nu_k \implies s_j \leq s_k$.

Proof. First note that, by the definition of subdifferentials $\partial P(x)$ for convex functions P, for any x, y in P's domain and $d_y \in \partial P(y)$ we have $x d_y - P(x) \leq y d_y - P(y)$. By the monotonicity property of subdifferentials for convex functions, we have for any $d_j \in \partial P(s_j)$ and $d_k \in \partial P(s_k)$ the inequality $(s_k - s_j)(d_k - d_j) \geq 0$. Thus, for $s_j < s_k$ we have $\min \partial P(s_k) \geq \max \partial P(s_j)$, yielding $\nu_j \leq s_j \min \partial P(s_k) - P(s_j) \leq s_k \min \partial P(s_k) - P(s_k) \leq \nu_k$. For $\nu_j < \nu_k$, let $d_j \in \partial P(s_j)$ and $d_k \in \partial P(s_k)$ satisfy Equation (3.5). We get $s_k d_j - P(s_k) \leq s_j d_j - P(s_j) < s_k d_k - P(s_k)$, which implies $d_j < d_k$. This, in turn, implies $s_j \leq s_k$ by monotonicity of subdifferentials. $\square$

Lemma 52. *For $\epsilon, \nu_0 > 0$ with $S(\nu_0) \neq \emptyset$, there exists $\delta > 0$ such that if $\nu \in [\nu_0 - \delta, \nu_0 + \delta]$ with $\mathcal{S}(\nu) \neq \emptyset$, then $\max S(\nu) - \min S(\nu) < \epsilon$ and there are $s_0 \in \mathcal{S}(\nu_0)$, $s \in S(\nu)$ with $s \in [s_0 - \epsilon, s_0 + \epsilon]$.*

Proof. For the sake of a contradiction, suppose the statements do not hold. Let us start with the second statement. So, there are $\epsilon, \nu_0 > 0$ such that for any $\delta > 0$ there is a $\nu \in [\nu_0 - \delta, \nu_0 + \delta]$ with $\mathcal{S}(\nu) \neq \emptyset$ and for all $s_0 \in \mathcal{S}(\nu_0)$ and $s \in S(\nu)$, $s \notin [s_0 - \epsilon, s_0 + \epsilon]$. We assume that for all δ it is possible to choose $s > \max S(\nu_0) + \epsilon$ (the other case is symmetric). Then we have $\nu > \nu_0$. Note that, by Lemma 51, this implies that for all $\nu' > \nu$, we have that $s' \geq s > \max S(\nu_0) + \epsilon$ for any $s' \in \mathcal{S}(\nu')$. Therefore, since ν' can be made arbitrarily close to ν by making δ arbitrarily small, for some $\tilde{s} \in (\max \mathcal{S}(\nu_0), s)$, there is no value $\tilde{\nu}$ where $\tilde{\nu} = \tilde{s}d - P(\tilde{s})$ for some $d \in \partial P(\tilde{s})$. However, by our restrictions on P, this is not possible, and thus this case cannot occur.

Now assume the first statement does not hold. That is, there are $\epsilon, \nu_0 > 0$ such that for any $\delta > 0$ there is a $\nu \in [\nu_0 - \delta, \nu_0 + \delta]$ with $\mathcal{S}(\nu) \neq \emptyset$ and $\max S(\nu) - \min S(\nu) \geq \epsilon$. Choose $\delta = 1$ and let ν be the affectance satisfying our assumption. We consider the case $\nu > \nu_0$ (the case $\nu < \nu_0$ is symmetric). Let $\delta' = (\nu - \nu_0)/2$ and let ν' be the affectance corresponding to δ'. Note that since $\max \mathcal{S}(\nu') - \min \mathcal{S}(\nu') \geq \epsilon$, it must be that for any $s_0 \in \mathcal{S}(\nu_0)$ and $s \in \mathcal{S}(\nu)$ we have $s - s_0 \geq \epsilon$. Similarly, we can define $\delta'' = (\nu_0 - \nu')/2$ to obtain ν'' and, by a similar argument, we get that for any $s_0 \in \mathcal{S}(\nu_0)$ and $s' \in \mathcal{S}(\nu')$ we have $s' - s_0 \geq \epsilon$. Together this implies $s - s_0 > 2\epsilon$. Repeating this k times, we obtain $s - s_0 > k\epsilon$. Thus s is larger than any finite number, yielding $\mathcal{S}(\nu) = \emptyset$, a contradiction to our assumption. □

Observation 53. *Consider a fixed ν and speeds $s_1 < s_2$ such that for $i \in \{1, 2\}$ there is $d_i \in \partial P(s_i)$ with $\nu = s_i d_i - P(s_i)$. Then P is differentiable in all $s \in (s_1, s_2)$ with $P'(s) = d_1 = \max \partial P(s_1) = d_2 = \min \partial P(s_2)$.*

Proof. Let $s \in (s_1, s_2)$. To see that P is differentiable at s, note that if $\partial P(s)$ contains more than one point, then there is a $\nu' \neq \nu$ such that for some $d \in \partial P(s)$ we have $\nu' = sd - P(s)$. If $\nu' < \nu$, this contradicts Lemma 51 (since $s > s_1$). For $\nu' > \nu$ we get a symmetric contradiction. Similar arguments show that $d_1 = \max \partial P(s_1)$ and $d_2 = \min \partial P(s_2)$. It remains to show that $P'(s) = d_1 = d_2$. Let $v_1, v_2 \in [s_1, s_2]$ with $v_1 < v_2$, and let $c_1 = \max \partial P(v_1)$ and $c_2 = \min \partial P(v_2)$. Then, if $c_1 \neq c_2$, the monotonicity of subdifferentials implies $c_1 < c_2$. And by the definition of subdifferentials we have $c_2 \geq \frac{P(v_2) - P(v_1)}{v_2 - v_1}$, which implies

$$c_2 v_2 - P(v_2) \geq c_2 v_1 - P(v_1) > c_1 v_1 - P(v_1).$$

Since both the left- and righthand side are equal to ν, this is a contradiction. $\square$

Observation 54. *For all $\nu, \beta, \epsilon > 0$ there exists $\delta > 0$ such that for $\beta' \in [\beta - \delta, \beta + \delta]$ we have $\max \mathcal{S}'(\nu) - \min \mathcal{S}'(\nu) < \epsilon$. Moreover, for some $s \in \mathcal{S}(\nu)$ and all $s' \in \mathcal{S}'(\nu)$ we have $|s - s'| < \epsilon$. Furthermore, $s > s'$ implies $\beta \leq \beta'$, and $s < s'$ implies $\beta \geq \beta'$.*

Proof. For the power function βP, Equation (3.5) becomes $\nu_j = s_j \beta d - \beta P(s_j)$ (for some $d \in \partial P(s_j)$). Thus, s' must be a solution to $\nu = s'\beta' d - \beta' P(s')$ for some $d \in \partial P(s')$, which is equivalent to a solution to $\beta\nu/\beta' = s'\beta d - \beta P(s')$. Since we can make β/β' arbitrarily close to 1 by making δ arbitrarily small, Lemma 52 gives a δ such that $\max \mathcal{S}'(\nu) - \min \mathcal{S}'(\nu) < \epsilon$ and $|s - s'| < \epsilon$. The last statement of the lemma follows from Lemma 51 (e.g., $s > s'$ implies $\nu \geq \beta\nu/\beta'$). $\square$

Lemma 55. *Consider two optimal flow plus energy schedules S and S' for β and β', respectively. Then, for every job j, $\beta < \beta'$ implies $\nu_j \leq \nu'_j$.*

Proof. For the sake of a contradiction, let j be the earliest released job for which the lemma's statement does not hold. Let i and i' be the job groups of j in S and S', respectively. If j is not the first job in J_i, let a be the job that completes before j. Then $\nu_a = \nu_j + 1$ and, by Lemma 56, $\nu'_a \leq \nu'_j + 1$, so $\nu_a > \nu'_a$, contradicting our choice of j. Thus j must be the first job in J_i. Note that s'_j satisfies $\nu'_j = s'_j \beta' d - \beta' P(s'_j)$ for some $d \in \partial P(s'_j)$. This implies $\beta\nu'_j/\beta' = s'_j \beta d - \beta P(s'_j)$ and, by $\nu_j > \nu'_j > \beta\nu'_j/\beta'$ and Lemma 51, $s'_j \leq s_j$. Thus, j completes in S not later than in S' (it runs at least as fast immediately at its release in S). So the second job in J_i also runs earlier in S than S'. Moreover, since it has affectance $\nu_j - 1$ in S and $\nu'_j - 1$ in S', the same argument shows that it runs not slower in S than in S'. By induction, we get that every job in J_i completes no later in S than in S'. Let k be the last job of J_i and note that $\nu_k > \nu'_k$. There are two cases:

Case 1: *k ends at the release of the next job run in S:* Let c be the job run after k in S. We can apply the same arguments as above to find a "new k" for c's job group. Since there are only a finite number of jobs, this can be repeated only a finite number of times, so that, eventually, we find a k for which the next case holds.

Case 2: *k ends before the release of the next job to run in S (or it is the last job):* This immediately yields a contradiction, since $1 = \nu_k > \nu'_k \geq 1$. □

Characterizing Optimal Schedules: We first prove that the affectance property characterizes optimal schedules. Lemma 56 shows that this property is necessary, Lemma 57 shows that affectance is unique across optimal schedules, and Corollary 59 shows that the affectance property is sufficient for optimality.

Lemma 56. *Any optimal schedule for* `FE-ICUU` *satisfies the affectance property.*

Proof. We prove the stronger fact that this lemma holds for optimal schedules for `FE-ICWA`. For the sake of a contradiction, suppose we have an optimal schedule that does not satisfy the affectance property. Then for some job group, either some combination of (3.3), (3.4), and (3.5) has no solutions, or $\mathcal{N}^i$ has no maximum. We first show that every combination of (3.3), (3.4), and (3.5) has some solution. We focus on showing that (3.4) and (3.5) together have some solution, and if that is the case then (3.3), (3.4), and (3.5) together also have some solution; It is straightforward to see that every other subset of (3.3), (3.4), and (3.5) must have some solution.

Suppose Equations (3.4) and (3.5) cannot be simultaneously satisfied. For each job j, let $\mathcal{N}^i_j$ be the set of values v^i for which Equations (3.4) and (3.5) are satisfied. Since $\partial P(s_j)$ is closed and convex,

$$\mathcal{N}^i_j = [s_j \min \partial P(s_j) - P(s_j) - u_j, s_j \max \partial P(s_j) - P(s_j) - u_j].$$

By assumption, Equations (3.4) and (3.5) cannot be simultaneously satisfied, so there are two jobs $j, k \in J_i$ with $\mathcal{N}^i_j \cap \mathcal{N}^i_k = \emptyset$ (otherwise $v^i \in \bigcap_{j \in J_i} \mathcal{N}^i_j \neq \emptyset$ satisfies (3.4) and (3.5)). Let $\eta_j = \max \mathcal{N}^i_j$ and $\eta_k = \min \mathcal{N}^i_k$ and assume, without loss of generality, $\eta_j < \eta_k$. Then, we have:

$$\eta_j + u_j = s_j \max \partial P(s_j) - P_j(s_j) \tag{3.6}$$

$$\eta_k + u_k = s_k \min \partial P(s_k) - P_k(s_k). \tag{3.7}$$

Since j and k are in the same job group, there is an $\epsilon > 0$ such that, if we decrease the running time of j by ϵ and increase the running time of k by ϵ, the flow changes by $\epsilon(u_k - u_j)$.

For $\gamma \leq \epsilon$, let $U(\gamma)$ be the cost change incurred by both, decreasing j's running time and increasing k's running time, by γ. With $p_j^* = p_j/s_j$ and $p_k^* = p_k/s_k$, j's speed after the change is $p_j/(p_j^* - \gamma)$ (and similar for k). We get

$$U(\gamma) = \gamma(u_k - u_j) + (p_j^* - \gamma)P\left(\frac{p_j}{p_j^* - \gamma}\right) - p_j^* P(s_j) + (p_k^* + \gamma)P\left(\frac{p_k}{p_k^* + \gamma}\right) - p_k^* P(s_k).$$

We show that $U'_{\uparrow}(0) < 0$ (derivative as γ increases), which contradicts the schedule's optimality. This derivative is given by

$$\begin{aligned} U'_{\uparrow}(\gamma) = u_k - u_j + \frac{p_j}{p_j^* - \gamma} \max \partial P\left(\frac{p_j}{p_j^* - \gamma}\right) - P\left(\frac{p_j}{p_j^* - \gamma}\right) \\ - \frac{p_k}{p_k^* + \gamma} \min \partial P\left(\frac{p_k}{p_k^* + \gamma}\right) + P\left(\frac{p_k}{p_k^* + \gamma}\right), \end{aligned}$$

which yields $U'_{\uparrow}(0) = u_k - u_j + s_j \max \partial P(s_j) - P(s_j) - s_k \min \partial P(s_k) + P(s_k)$. By applying Equations (3.6) and (3.7), we obtain the desired contradiction: $U'_{\uparrow}(0) = \eta_j - \eta_k < 0$.

Now suppose Equations (3.4) and (3.5) can be simultaneously satisfied, but not together with Constraint (3.3). We show one useful fact first. Note that, for any job j in any schedule, the derivative of the objective as speed increases (assuming the order in which jobs are run remains fixed) is

$$V'_{\uparrow}(s_j) = -u_j p_j / s_j^2 + (p_j \max \partial P(s_j))/s_j - p_j P(s_j)/s_j^2$$

and the derivative of the objective as speed decreases is

$$V'_{\downarrow}(s_j) = -l_j p_j / s_j^2 + (p_j \min \partial P(s_j))/s_j - p_j P(s_j)/s_j^2.$$

Observe that if $V'_{\uparrow}(s_j) < 0$, then there exists some $\epsilon > 0$ such that if we increase the speed of j by ϵ, the total objective decreases. The same is true for $V'_{\downarrow}(s_j) > 0$ and decreasing the speed. Thus, since the schedule is optimal, we have

$$V'_{\uparrow}(s_j) \geq 0 \text{ and } V'_{\downarrow}(s_j) \leq 0. \tag{3.8}$$

Let $\mathcal{M}^i = \bigcap_{j \in J_i} \mathcal{N}_j^i$ denote the set of solutions ν^i to Equations (3.4) and (3.5). As a finite intersection of closed and convex sets, $\mathcal{M}^i$ itself is closed and convex. Together with

$\mathcal{M}^i \cap [0, l_{\ell(i)} - u_{\ell(i)}] = \emptyset$, this implies either $\iota = \max \mathcal{M}^i < 0$ or $\psi = \min \mathcal{M}^i > \zeta_{i+1}(\nu^{i+1} + W_{i+1})$. In the first case, there is a job j such that $\iota + u_j = s_j \max \partial P(s_j) - P(s_j)$. This implies

$$V'_{\uparrow}(s_j) = -u_j/s_j^2 + \max \partial P(s_j)/s_j - P(s_j)/s_j^2 = \iota/s_j^2 < 0,$$

a contradiction to Equation (3.8). In the second case, there is a job j such that $\psi + u_j = s_j \min \partial P(s_j) - P(s_j)$. We consider two subcases. The easier case is $\psi > l_{\ell(i)} - u_{\ell(i)} = l_j - u_j$, because then we have

$$0 = -u_j/s_j^2 + \min \partial P(s_j)/s_j - P(s_j)/s_j^2 - \psi/s_j^2 < -l_j/s_j^2 + \min \partial P(s_j)/s_j - P(s_j)/s_j^2 = V'_{\downarrow}(s_j),$$

a contradiction to Equation (3.8). For the second subcase, suppose $\nu^{i+1} + W_{i+1} < \psi \leq l_{\ell(i)} - u_{\ell(i)}$. Let κ be the smallest index of a job group such that (a) j lower affects jobs in κ, and (b) J_κ contains k with $\nu^\kappa + u_k = s_k \max \partial P(s_k) - P(s_k)$. Such a job group exists, as otherwise ν^{i+1} would be larger. By our choice of κ, we have $\nu^y = \nu^{y+1} + W_{y+1}$ for all $y \in \{\, i+1, \ldots, \kappa - 1 \,\}$ and thus $\nu^\kappa + \sum_{x=i+1}^{\kappa} W_x < \psi$. We have the equations

$$\begin{aligned} \psi + u_j &= s_j \min \partial P(s_j) - P_j(s_j) \\ \nu^\kappa + u_k &= s_k \max \partial P(s_k) - P_k(s_k). \end{aligned}$$

Note that $u_j = l_j - (l_j - u_j)$ and, since j lower affects k, $(l_j - u_j) - (l_k - u_k) = \sum_{x=i+1}^{\kappa} W_x$. Therefore, we can write

$$\psi + l_j - \left(l_k - u_k + \sum_{x=i+1}^{\kappa} W_x \right) = s_j \min \partial P(s_j) - P(s_j) \tag{3.9}$$

$$\nu^\kappa + l_k - (l_k - u_k) = s_k \max \partial P(s_k) - P(s_k). \tag{3.10}$$

Since j lower affects k, there is an $\epsilon > 0$ such that, if we increase the running time of j by ϵ and decrease the running time of k by ϵ, the flow changes by $\epsilon(l_j - u_k)$. Similar to above, we can define $\widetilde{U}(\gamma)$ for $\gamma \leq \epsilon$ to be the cost change incurred by both, decreasing j's running time and increasing k's running time, by γ. With $p_j^* = p_j/s_j$ and $p_k^* = p_k/s_k$, we once more can compute the cost change:

$$\widetilde{U}(\gamma) = \gamma(l_j - l_k) + (p_j^* + \gamma)P\left(\frac{p_j}{p_j^* + \gamma}\right) - p_j^* P(s_j) + (p_k^* - \gamma)P\left(\frac{p_k}{p_k^* - \gamma}\right) - p_k^* P(s_k).$$

We show that $U'_{\uparrow}(0) < 0$ (derivative as γ increases), which contradicts the schedule's optimality. The derivative is given by

$$\widetilde{U}'_{\uparrow}(\gamma) = l_j - l_k - \frac{p_j}{p^*_j + \gamma} \min \partial P\left(\frac{p_j}{p^*_j + \gamma}\right) + P\left(\frac{p_j}{p^*_j + \gamma}\right) + \frac{p_k}{p^*_k - \gamma} \max \partial P\left(\frac{p_k}{p^*_k - \gamma}\right) - P\left(\frac{p_k}{p^*_k - \gamma}\right),$$

which yields $\widetilde{U}'_{\uparrow}(0) = l_j - l_k - s_j \min \partial P(s_j) + P(s_j) + s_k \max \partial P(s_k) - P(s_k)$. By applying Equations (3.9) and (3.10), we obtain the desired contradiction:

$$\widetilde{U}'_{\uparrow}(0) = l_j - l_k - s_j \min \partial P(s_j) + P(s_j) + s_k \max \partial P(s_k) - P(s_k) = \nu^{\kappa} + \sum_x W_x - \psi < 0.$$

The fact that $\mathcal{N}^i$ has a maximum follows from the facts that the subdifferential of a function at any point is closed, and that there are only a finite number of jobs. □

Lemma 57. *Let S_1 and S_2 be schedules with the affectance property and let ν^i_j denote the affectance of job j in the corresponding schedule. Then $\nu^1_j = \nu^2_j$ for all j.*

Proof. Suppose this were not the case. Let j be the job with the earliest release such that $\nu^1_j \neq \nu^2_j$. Without loss of generality, assume $\nu^1_j < \nu^2_j$. Let k be the job that completes before j. First note that j must be the first job in its job group in at least one of the schedules. Otherwise, by Lemma 56, j would have affectance $\nu^i_j = \nu^i_k - 1$ in both schedules. This would yield $\nu^1_k \neq \nu^2_k$, a contradiction to our choice of j. In fact, j must begin at its release time at least in S_2: Otherwise, if j begins at its release in S_1 but not in S_2, we have $\nu^2_k = \nu^2_j + 1$. But Lemma 56 gives $\nu^1_k \leq \nu^1_j + 1$, implying the contradiction $\nu^2_j \leq \nu^1_j$.

So we have that j begins at its release in S_2 and either at or after its release in S_1. Let ι_1 and ι_2 be the job groups of j in the respective schedules and $\mathbf{J} = \mathbf{J}_{\iota_1} \cap \mathbf{J}_{\iota_2}$. By Lemma 56, every job in $\mathbf{J}$ has smaller affectance in S_1 than in S_2. Thus, by Lemma 51, every job in $\mathbf{J}$ runs in S_1 not faster than in S_2. Let κ be the last job of $\mathbf{J}$. Then κ does not complete later in S_2 than in S_1. We distinguish two cases:

Case 1: *κ completes at the same time in S_1 and S_2:* Then, every job in $\mathbf{J}$ runs at the same speed in S_1 and S_2. If κ does not complete at the release of another job, we have $1 = \nu_\kappa^1 \neq \nu_\kappa^2 = 1$, a contradiction. Equations (3.4) and (3.5) will have the same set of feasible solutions in both schedules. Thus, the only reason to have $\nu_j^1 < \nu_j^2$ is that Constraint (3.3) forces it. Let ρ be the job run after κ. Note that in any optimal schedule and any job group i, $\zeta^i(\nu^i + W_i)$ is the affectance of the first job run in that job group. Therefore, since $\zeta^{\iota_1+1}(\nu^{\iota_1+1} + W_{\iota_1+1})$ in S_1 is smaller than $\zeta^{\iota_2+1}(\nu^{\iota_2+1} + W_{\iota_2+1})$ in S_2, we must have $\nu_\rho^1 < \nu_\rho^2$. We can apply the same arguments as above to find a "new κ" for ρ's job group. Since there are only a finite number of jobs, this can be repeated only a finite number of times, so that, eventually, we find a κ for which the next case holds.

Case 2: *κ completes later in S_1 than in S_2:* There are two possibilities for how κ completes in S_2:

2.a: *κ ends at the release of the next job run in S_2:* Let ρ be the next job run and note that ρ runs after its release in S_1. This implies $\rho \in J_{\iota_1}$. Also note that we have $\nu_\rho^2 > \nu_\rho^1$ (since $\nu_\rho^2 \geq \nu_\kappa^2 - 1 > \nu_\kappa^1 - 1 = \nu_\rho^1$). Similar to above, we can iterate our arguments for ρ until, eventually, the next case holds.

2.b: *κ ends before the release of the next job in S_2, or κ is the last job:* In this case, Lemma 56 gives $\nu_\kappa^2 = 1$. Bit since we also have $\nu_\kappa^1 \geq 1$, this contradicts $\nu_\kappa^2 > \nu_\kappa^1$. □

Next, we show how to transform any schedule that has the affectance property into any other such schedule without changing the flow plus energy value. Together with Lemma 56, this immediately implies that the affectance property is sufficient for optimality (Corollary 59). Also, Lemma 56 is a nice algorithmic tool, as it allows us to find schedules "in between" any two optimal schedules with arbitrary precision. We will make use of that in the proof of Theorem 62.

Lemma 58. *Let S_1 and S_2 be schedules with the affectance property. Then S_1 can be transformed into S_2 without changing its flow plus energy value. All intermediate schedules satisfy the affectance property and we can make the speed changes between intermediate schedules arbitrarily small.*

Proof. Let S_1 and S_2 be two schedules with the affectance property. Define a *job section* to be a maximal set of jobs beginning at the same time in both schedules and with the property that all but the last job in the section end after the release time of the next job run in either S_1 or S_2. Note that, by Lemma 57, the affectance of each job in S_1 and S_2 is the same. Thus we do not distinguish between schedules when discussing affectance. For any pair of jobs j and j' that are in the same job section with j' running after j, we have $\nu_j = \nu_{j'} + 1$ (by Lemma 56 and because j and j' are in the same job group in S_1 or S_2).

To prove the lemma, we repeatedly modify S_1 such that, after each modification, either one additional job runs at the same speed in both schedules or one additional job section ends at the same time in both schedules. As there are only a finite number of jobs and job sections, this process will terminate. So let j be the earliest released job that is run at speeds $s_j^1 \neq s_j^2$ in S_1 and S_2, respectively. Without loss of generality, assume $s_j^1 > s_j^2$. Let i be the job section that j is in and j' the job run after j. We consider two cases:

Case 1: *Every job in i ends earlier in S_1 than in S_2:* We modify S_1 by decreasing the speed of j until either it runs at the same speed in S_1 and S_2, or the job section i ends at the same time in both S_1 and S_2. Note that the resulting schedule is feasible (we increase the starting times of jobs, but the ending time of the last job in i is at most the next job's release time). Let $\tilde{s}_j^1$ be the new speed of j in the modified S_1 schedule. Since the last job of i ends earlier in S_1, no job is run immediately after i in S_1. Thus, by Constraint (3.3) and Equation (3.4), we have $\nu_j = l_j$. Recall the objective change $V'_{\downarrow}(s_j)$ from the proof of Lemma 56, which is

$$V'_{\downarrow}(s_j) = -l_j p_j / s_j^2 + p_j \min \partial P(s_j)/s_j - p_j P(s_j)/s_j^2.$$

This yields $V'_{\downarrow}(s_j) = 0 \iff l_j = \nu_j = s_j \min \partial P(s_j) - P(s_j)$. By Observation 53, this holds for all $s_j \in (s_j^2, s_j^1]$. Thus, by changing the speed from s_j^1 to any speed in (s_j^2, s_j^1), the objective does not change.

Case 2: *Some job in i ends at the same time or later in S_1 than S_2:* Then there must be a job in i that runs faster in S_2 than in S_1. Let k be the first such job. We modify S_1 by decreasing the speed of j and increasing the speed of k until either j runs at the same speed as in S_2 or k runs at the same speed as in S_2. We modify the speeds such that the completion time of j' does not change. Thus, the number of jobs whose flows change is $l_j - l_k$. Since, furthermore, the times when we start jobs between j and k is only increasing, the resulting schedule is feasible. Similar to the proof Lemma 56, $U(\gamma)$ be the cost change incurred by both, increasing j's running time and decreasing k's running time, by γ. Then, the objective change rate is

$$\begin{aligned} U'_{\uparrow}(\gamma) = l_j - l_k - \frac{p_j}{p_j^* + \gamma} \min \partial P\left(\frac{p_j}{p_j^* + \gamma}\right) + P\left(\frac{p_j}{p_j^* + \gamma}\right) \\ + \frac{p_k}{p_k^* - \gamma} \max \partial P\left(\frac{p_k}{p_k^* - \gamma}\right) - P\left(\frac{p_k}{p_k^* - \gamma}\right), \end{aligned}$$

where $p_j^* = p_j/s_j^1$ and $p_k^* = p_k/s_k^1$. By Observation 53, we have for any γ with $p_j/(p_j^*+\gamma) \in (s_j^2, s_j^1]$ and $p_k/(p_k^* - \gamma) \in [s_k^1, s_k^2)$ the identities $\nu_j = \frac{p_j}{p_j^*+\gamma} \min \partial P\big(\frac{p_j}{p_j^*+\gamma}\big) - P\big(\frac{p_j}{p_j^*+\gamma}\big)$ and $\nu_k = \frac{p_k}{p_k^*-\gamma} \max \partial P\big(\frac{p_k}{p_k^*-\gamma}\big) - P\big(\frac{p_k}{p_k^*-\gamma}\big)$. Because of $l_j - l_k = \nu_j - \nu_k$, we have $U'_{\uparrow}(\gamma) = 0$ for all such γ. That is, the modification of the schedule does not change the objective.

The lemma's last part follows immediately from this, as when we increase/decrease jobs' running times, we can change the running time by a smaller value than we actually did above, without changing the objective or the adherence to the affectance property. □

Corollary 59. *Any schedule satisfying the affectance property is optimal.*

Binary Search Algorithm: We now provide the main technical result of this section, a polynomial time algorithm for `B-ICUU` based on any such algorithm for `FE-ICUU` (Theorem 62). In order to state the algorithm and its correctness, we need two more auxiliary lemmas. Lemma 60 proves that the affectance of jobs is continuous in β, while Lemma 61 does the same for job speeds.

Lemma 60. *For $\beta > 0$ and $\epsilon > 0$, there exists $\delta > 0$ such that for all jobs j and $\beta' \in [\beta-\delta, \beta+\delta]$, any optimal `FE-ICCU` schedules S for β and S' for β' adhere to $\nu'_j \in [\nu_j-\epsilon, \nu_j+\epsilon]$.*

Proof. First note that, since there are only finitely many jobs, it is sufficient to prove the lemma for each job separately. Now, for the sake of a contradiction, assume the statement does not hold for a fixed job j. Then, there are $\beta, \epsilon > 0$ such that for any $\delta > 0$ there is a $\beta' \in [\beta - \delta, \beta + \delta]$ with $|\nu_j - \nu'_j| > \epsilon$. Let j be the first such job. We distinguish two cases, depending on whether ν_j or ν'_j is larger. In the following, we show how to handle the case

$\nu_j - \nu'_j > \epsilon$; the other one is symmetric. By Lemma 55, we have $\beta > \beta'$. Let i and i' be the job groups of j in S and S', respectively. Note that we can choose δ small enough such that all jobs completing before j have their affectance change by less than ϵ. Thus, the job that completes before j cannot be in J_i, since otherwise, by Lemma 56, the change in affectance for that job would be larger than ϵ. Additionally, by Lemma 52, we can choose δ small enough such that j begins running in S' at most γ later than it does in S ($\gamma > 0$ to be chosen later). Our goal is to find some job such that Constraint (3.3) is tight on that job, showing that it finishes earlier in S than in S' (which will, eventually, lead to a contradiction). We consider jobs in order of release time, starting with j. Let a be the job we are currently considering. If a is not run immediately after the previous job completed, we have a contradiction since $1 = \nu_a > \nu'_a + \epsilon = 1$. There are two cases to consider:

Case 1: *a is in the same job group as j:* Let k be the number of jobs run after j until a completes. This yields $\nu_a = \nu_j - k$ and $\nu'_a = \nu'_j - k$. If $\nu'_a = s'_a\beta' \max \partial P(s'_a) - P(s'_a)$, we are done (see below). Otherwise, $\nu^{i'}$ is not constrained by the speed of job a (i.e., some other job must be preventing $\nu^{i'}$ from being larger). For small enough δ and for some $d \in \partial P(s'_a)$, we have $\nu_a > \nu'_a + \epsilon > \beta\nu'_a/\beta' = s'_a\beta d - \beta P(s'_a)$. We get $s_a \geq s'_a$, so a finishes in S not later than in S'.

Case 2: *a is not in the same job group as j:* We have shown that $\nu^{i'}$ is not constrained by Equations (3.4) and (3.5). Thus, it must be constrained by Equation (3.3), yielding $\nu'_a = \nu^{i'}$. Additionally, $\nu_a \geq \nu^i$, and thus the properties that held for j also hold for a, so we can take a as the "new j" and repeat the same arguments.

Let b be the job such that $\nu'_b = s'_b\beta' \max \partial P(s'_b) - \beta' P(s'_b)$. For small enough δ we have $\nu_b > \nu'_b + \epsilon > \beta\nu'_b/\beta' = s'_b\beta \max \partial P(s'_b) - \beta P(s'_b)$ implying $\nu_b - \epsilon > s'_b\beta \max \partial P(s'_b) - \beta P(s'_b)$. By Lemma 52, there is some $\tilde{s} \in (s'_b, s_b]$ which is a solution to

$$s'_b\beta \max \partial P(s'_b) - \beta P(s'_b) + \epsilon = \tilde{s}\beta \max \partial P(\tilde{s}) - \beta P(\tilde{s}).$$

Since $s_b \geq \tilde{s} > s'_b$, we have that b completes strictly earlier in S than in S'. In a manner similar to previous results, we now achieve a contradiction: Let c be the last job in the job group of b in S. We consider two cases:

Case 1: *c ends at the release of the next job run in S:* Let x be the job run after c in S. Note that the same conditions hold for x as those that held for b, in that x is run at its release and $\nu_x \geq \nu_c - 1 > \nu'_c + \epsilon - 1 = \nu'_x$. Thus we can repeat the above arguments to find a "new b". Since there are only a finite number of jobs, this can be repeated only a finite number of times until c either ends before the release of the next job run in S, or c is the last job.

Case 2: *c ends before the release of the next job to run in S, or it is the last job:* Here we have reached a contradiction, since $1 = \nu_c > \nu'_c + \epsilon \geq 1$. □

Lemma 61. *For $\beta > 0$ and $\epsilon > 0$, there exists $\delta > 0$ such that for all jobs j and $\beta' \in [\beta - \delta, \beta + \delta]$, any optimal* `FE-ICUU` *schedules S for β and S' for β' adhere to $s'_j \in [s_j - \epsilon, s_j + \epsilon]$.*

Proof. For the sake of a contradiction, assume the statement does not hold. That is, there are $\beta, \epsilon > 0$ such that for any $\delta > 0$ there is some $\beta' \in [\beta - \delta, \beta + \delta]$ and some job j such that there are S and S' with $s'_j \notin [s_j - \epsilon, s_j + \epsilon]$. Let j be the first such job and let i be its job group in S. We consider only case $s'_j > s_j + \epsilon$, the other one is symmetric.

Note that the completion time of j decreases for any β' satisfying our assumptions. Let $\gamma > 0$ and $k \in J_i$ released after j be such that for every job $j' \in J_i$ processed between j and k, j' completes at least γ earlier in S' than in S, and further no such γ exists for k. We will later distinguish the cases whether k exists or not. Let p^*_j be the processing time of j in S. Note that j has the highest affectance of any job in J_i after j, and thus must have the highest speed (Lemma 51). Thus for j, it takes the largest speed increase Δs_j to decrease its running time by γ, which is $\Delta s_j = p_j/(p^*_j - \gamma) - p_j/p^*_j$. By Lemma 52 and for a small enough δ_1, for $\tilde{\nu}_j \in [\nu_j - \delta_1, \nu_j + \delta_1]$ there is some $s \in \mathcal{S}(\nu_j)$ such that for any $\tilde{s} \in \mathcal{S}(\tilde{\nu}_j)$, $|\tilde{s} - s| < \Delta s_j/4$. By Lemma 60, there is some δ_2 such that if $\bar{\beta} \in [\beta - \delta_2, \beta + \delta_2]$, then $|\bar{\nu}_j - \nu_j| < \min\{\delta_1, 1\}$. Additionally, by Observation 54, for any $\tilde{\nu}_j \in [\nu_j - \delta_1, \nu_j + \delta_1]$ and small enough δ_3, for any $\beta^* \in [\beta - \delta_3, \beta + \delta_3]$ there is some $\tilde{s} \in \mathcal{S}(\tilde{\nu}_j)$ such that for all $s^* \in \mathcal{S}(\nu^*_j)$ we have $|\tilde{s} - s^*| < \Delta s_j/4$. Thus, by taking $\delta_4 = \min\{\delta_2, \delta_3\}$ we have that there is some $s \in \mathcal{S}(\nu_j)$ such that for all $s' \in \mathcal{S}(\nu'_j)$ we have $|s - s'| < \Delta s_j/2$.

Note that any job in J_i between j and k is lower affected by j in both S and S', since for any job in J_i after j, it begins earlier in S', and if it ceased to be lower affected in S', the affection of j would decrease by at least 1. Let $0 < \gamma' < \gamma$. Thus, in a schedule for energy function P_β where jobs in J_i between j and k completed γ' earlier, their upper affections would not change. But then, their lower affection and upper affection must be equal, and so their affectance would not change. Thus if all jobs in J_i between j and k complete γ' earlier, they are running at the same speed as they do in S. There are two cases, based on the existence of k:

Case 1: *k does not exist.* As we have seen, by taking $\delta = \delta_4$, there is some $\bar{s}_j \in \mathcal{S}(\nu_j)$ such that if j runs at that speed, it completes γ' earlier. Thus, the schedule that is idential to S except that j runs at $\bar{s}_j$ satisfies the affectance property But then, by Corollary 59, it is optimal, contradicting our assumption that such a γ exists.

Case 2: *k does exist.* For job k, since such γ does not exist for it, we can choose δ small enough such that it ends arbitrarily close to the same time, or later, in S' than S. We will choose δ_5 in a similar manner to above such that there is a speed where k takes γ' longer to complete than it does in S. More formally, let p^*_k be the processing time of k in S. Let Δs_k be the speed decrease of k required for it to take γ' longer to complete than in S. By Lemma 52, for a small enough δ_6, if $\tilde{\nu}_k \in [\nu_k - \delta_6, \nu_k + \delta_6]$, then there is some $s \in \mathcal{S}(\nu_k)$ such that for any $\tilde{s} \in \mathcal{S}(\tilde{\nu}_k)$, $|\tilde{s} - s| < \Delta s_k/4$. By Lemma 60, there is some δ_7 such that if $\bar{\beta} \in [\beta - \delta_7, \beta + \delta_7]$, then $|\bar{\nu}_k - \nu_k| < \min\{\delta_6, 1\}$. Additionally, by Observation 54, for any $\tilde{\nu}_k \in [\nu_k - \delta_6, \nu_k + \delta_6]$ and small enough δ_8 it holds that for any $\beta^* \in [\beta - \delta_8, \beta + \delta_8]$, there is some $\tilde{s} \in \mathcal{S}(\tilde{\nu}_k)$ such that for all $s^* \in \mathcal{S}(\nu^*_k)$ we have $|\tilde{s} - s^*| < \Delta s_k/4$. Thus by taking $\delta_5 = \min\{\delta_7, \delta_8\}$ we have that there is some $s \in \mathcal{S}(\nu_k)$ such that for all $s' \in \mathcal{S}(\nu'_k)$ we have $|s - s'| < \Delta s_k/2$. Thus, it must be that $\Delta s_k \in \mathcal{S}(\nu_k)$. By taking $\delta = \min\{\delta_4, \delta_5\}$, we have that there are speeds $\bar{s}_j \in \mathcal{S}(\nu_j)$ and $\bar{s}_k \in \mathcal{S}(\nu_k)$ such that if we modify S such that j runs at $\bar{s}_j$ and k runs at $\bar{s}_k$, j ends γ' earlier and k ends γ' later, thus the schedule after k does not change, and the new schedule satisfies the affectance property. Thus, by Corollary 59, it is optimal, contradicting our assumption that such a γ exists. □

Theorem 62. *Given a polynomial time algorithm for the continuous flow plus energy problem with unit size unit weight jobs, there is a polynomial time algorithm for the budget variant.*

Proof. Suppose we are given an energy budget B, and an algorithm to solve `FE-ICUU`. As we formally show in the proof of Theorem 64, the energy of optimal schedules increases as β decreases (even though we are considering here integral flow rather than fractional flow). Thus, the first step of the algorithm is to binary search over β until we find a schedule that fully utilizes B. If we find such a β, we are done (any optimal `FE-ICUU` schedule must minimize flow for the energy it consumes). Otherwise, we consider three cases:

Case 1: We find a β for which the optimal `FE-ICUU` schedule runs every job at the lowest speed used by any optimal schedule and uses $> B$ energy. Here, this lowest speed is (if it exists) the largest speed s such that for all $s' < s$ we have $\frac{P(s)}{s} \leq \frac{P(s')}{s'}$. In this case, no solution exists, since running a job at a lower speed increases its flow but does not decrease its energy.

Case 2: We find a β for which the optimal `FE-ICUU` schedule runs every job at the highest speed used by any optimal schedule and uses $\leq B$ energy. Here, this highest speed is (if it exists) the largest speed s such that for all $s' > s$ we have $P(s') = \infty$. In this case, β yields the optimal budget solution, since running any job at a higher speed uses infinite energy.

Case 3: There is $\epsilon > 0$ such that for any β, the computed optimal `FE-ICUU` schedule uses at least $B + \epsilon$ or at most $B - \epsilon$ energy. Since job speeds are continuous in β (Lemma 61) and the energy increases as β decreases, we know that there is some β such that the corresponding `FE-ICUU` solutions contain schedules using both $B + \epsilon_1$ energy and $B - \epsilon_2$ energy ($\epsilon_1, \epsilon_2 > 0$). Fix such a β and let S_1 and S_2 be the corresponding schedules using $B - \epsilon_1$ and $B + \epsilon_2$, respectively. By Lemma 58, we can continuously change the speeds (and, thus, energy) of S_1 to obtain S_2. During this process, we obtain an intermediate optimal `FE-ICUU` schedule that uses exactly B energy. As described above, this schedule is also optimal for `B-ICUU`. □

3.4.2 Reduction from `*-*D**` to `*-*C**`

The main result of this subsection is a reduction from the discrete to the continuous setting. Using mild computational power assumptions, Theorem 63 shows how to use an algorithm for the continuous variant of one of our problems (`*-*C**`) to solve the corresponding discrete variant (`*-*D**`).

Theorem 63. *Given a polynomial time algorithm for any budget or flow plus energy variant in the continuous setting, there is a polynomial time algorithm for the corresponding discrete variant.*

Proof of Theorem 63. Consider a discrete problem with speeds $s_0 = 0 \leq s_1 \leq s_2 \leq \cdots \leq s_n$ and powers $0 \leq P_0 = P_1 \leq \cdots \leq P_n$. Define an interpolated power function $P\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ for $s \in [s_i, s_{i+1}]$ by $P(s) = P_i + \gamma(P_{i+1} - P_i)$, where $\gamma \in [0,1]$ such that $s = s_i + \gamma(s_{i+1} - s_i)$; and $P(s) = \infty$ for $s > s_n$. For this power function, running in the continuous setting for time T at speed s is equivalent to running in the discrete setting for time $(1-\gamma)T$ at speed s_i and for time γT at speed s_{i+1}. W.l.o.g., we can assume $\frac{P_{i+1}-P_i}{s_{i+1}-s_i} > \frac{P_i - P_{i-1}}{s_i - s_{i-1}}$. Otherwise, we could save energy by interpolating s_i with s_{i-1} and s_{i+1}. The resulting function P is non-decreasing, continuous, and convex. This allows us to apply the algorithm for the continuous setting to compute a schedule. Using interpolation, any such schedule can be transformed into an equivalent schedule for the discrete speeds in polynomial time. □

3.4.3 Reducing from `B-FC**` to `FE-FC**`

This subsection gives a reduction from the budget to the flow plus energy objective. The reduction given in Theorem 64 is for fractional flow, assumes the most general setting (weighted jobs of arbitrary size), and preserves unit size and unit weight jobs, making it applicable to reduce `B-FC**` to `FE-FC**`. The key idea is that if we consider the behavior of flow plus β energy and as we vary energy, the resulting function has a single local minimum. Therefore, using a local search technique we will be able to converge to the energy budget that corresponds to the minimum flow plus β energy.

Theorem 64. *Given a polynomial time algorithm for the budget variant and fractional flow, there is a polynomial time algorithm for the corresponding flow plus energy variant.*

Proof. Consider an instance of the `B-FCWA` problem with energy budget B and assume we are given an algorithm for the `FE-FCWA` problem. Note that if we can find a β such that the minimum flow plus β time energy schedule S returned by the FE-FCWA algorithm uses exactly B energy, then S is optimal for the B-FCWA instance. To efficiently find such a β, we argue that the energy of the minimum flow plus energy schedule is both decreasing and continuous in β. The theorem's statement then follows via binary search over β.

Continuity follows from a result in Chapter /refchp:green. To see that energy is decreasing in β, let us assume this is not the case and derive a contradiction. So there are $\beta_1 < \beta_2$ and two corresponding minimum flow plus energy schedules S_1 and S_2. Let F_1 and E_1 be the flow and energy of S_1. Similarly, F_2 and E_2 are the flow and energy of S_2. By our assumption we have $E_1 < E_2$. From the schedules' optimality, we know that $F_1 + \beta_1 E_1 \leq F_2 + \beta_1 E_2$ as well as $F_2 + \beta_2 E_2 \leq F_1 + \beta_2 E_1$. Solving each inequality for the corresponding β value yields

$$\beta_2 \leq \frac{F_2 - F_1}{E_1 - E_2} \leq \beta_1,$$

a contradiction to our choice of β. □

3.4.4 Reducing from FE-FC** to B-FC**

We now show a reduction from FE-FC** to B-FC**. The key idea is that if we consider the behavior of flow plus β energy and as we vary energy, the resulting function has a single local minimum. Therefore, using a local search technique we will be able to converge to the energy budget that corresponds to the minimum flow plus β energy.

Lemma 65. *Given a polynomial time algorithm for B-FC**, there is a polynomial time algorithm for FE-FC**.*

Proof. For some fixed β let the optimal flow plus β energy schedule be S_1 using flow F_1 and energy E_1. We show that flow plus β energy is decreasing for optimal energy budget schedules with $B < E_1$ and increasing for optimal energy budget schedules with $B > E_1$. As noted in the proof of Theorem 64, the energy of optimal flow plus energy schedules is continuous and decreasing in β, thus for every energy E, there is a unique β such that the the minimum flow plus β energy schedule uses energy E, and the value of β is decreasing as energy increases.

We show that the energy is increasing for $B > E_1$, the argument for energy increasing for $B < E_1$ is symmetrical. Consider some energies $E_3 > E_2 > E_1$ and let S_3, S_2 be the corresponding minimum flow schedules with energy budgets E_3 and E_2. As noted before, this implies there exists some $\beta_3, \beta_2 < \beta_1$ such that the minimum flow plus β_2 energy schedule is S_2 and the minimum flow plus β_3 energy schedule is S_3. Now let β be the value such that

$$F_1 + \beta E_1 = F_2 + \beta E_2$$

Note that for all $\beta' > \beta$, $F_2 + \beta' E_2 < F_3 + \beta' E_3$ and for all $\beta' < \beta$, $F_3 + \beta' E_3 < F_2 + \beta' E_2$. This means that $\beta < \beta_1$, otherwise S_2 would not be the optimal schedule at $\beta_2 < \beta$. However this gives that $F_2 + \beta_1 E_2 < F_3 + \beta' E_3$ as desired. □

3.4.5 Reduction from FE-ICUU to B-ICUU

We now show a reduction from `FE-ICUU` to `B-ICUU`. The proof is almost identical to that of the previous subsection, with the only difference being that a new proof that β changes continuously in the energy budget is required.

Lemma 66. *Given a polynomial time algorithm for `B-ICUU`, there is a polynomial time algorithm for `FE-ICUU`.*

Proof. The fact that β must decrease as the energy budget increases (for budgets that map to optimal schedules) follows from arguments similar to those in the proof of Theorem 64. We must also show that each energy budget corresponds to an optimal flow plus energy schedule, and that β is continuous in the energy budget. This follows from Lemmas 58 and 61, since they together say that for any β that maps to multiple schedules, we can find a schedule using any energy between the minimum and maximum of such schedules, and that for any two β that are arbitrarily close, there will be optimal schedules for those values of β that are arbitrarily close in energy usage.

The remainder of the proof follows as the proof of Lemma 65. □

4.0 SPEED SCALING WITH A SOLAR CELL

Many devices, most notably sensors in hazardous environments, contain energy harvesting technologies. Solar cells are probably the most common example, but some sensors also harvest energy from ambient vibrations or electromagnetic radiation (e.g., from communication technologies such as television transmitters). To get a rough feeling for the involved scales, note that batteries can store on the order of a Joule of energy per cubic millimeter, while solar cells provide several micro-Watts per square millimeter in bright sunlight, and both vibrations and ambient radiation technologies provide on the order of nano-Watts per cubic millimeter. This Chapter considers an algorithmic problem related with to the power management of such devices. In particular, we study the following setting:

- The device has a speed-scalable processor that can run at any of a finite number of speeds $s_1 < \cdots < s_k$, each associated with a power consumption rate $P_1 < \cdots < P_k$. By time multiplexing, a job may be effectively run at any speed $s \in [0, s_k]$.
- The device harvests energy from its environment. For simplicity, we assume that it harvests energy at a time-invariant rate $R > 0$ (like a solar-cell in bright sunlight).
- The device has a battery to store harvested energy. For simplicity we assume that the capacity of the battery isn't a limiting factor.

The processor must process a collection of n jobs of various sizes and with associated time intervals that represent when the job arrives and when it has to be finished.

A variation of this setting, in which the allowable speeds were the non-negative reals and where the power was assumed to be a monomial function of the speed, was first considered in [8]. There, the authors studied also the objective of minimizing the recharge rate (i.e., the minimal rate at which to harvest energy in order to have sufficient energy to schedule all jobs). They showed that the offline problem can be expressed as a convex program. Thus, in principle the problem is solvable in polynomial-time, and the KKT conditions can be used to develop an algorithm that recognizes optimal solutions. Moreover, [8] also proved that the schedule that optimizes the total energy usage is a 2-approximation for the objective of recharge rate. Finally, they showed that the online algorithm BKP, which is known to be $O(1)$-competitive for total energy usage [6], is also $O(1)$-competitive with respect to the recharge rate. So, intuitively, the main take-away point from [8] was that schedules that naturally arise when minimizing energy usage are $O(1)$ approximate with respect to the recharge rate. However, [8] noted that "computing the recharge-rate optimal schedule is still seemingly much harder than computing an energy optimal schedule...". In particular, the latter has a particularly simple structure and can easily be computed in polynomial time [34]. A relatively recent survey of speed scaling and other energy management algorithmics can be found in [1].

While, in principle, a polynomial-time linear programming algorithm can solve this problem, this is unsatisfying for two reasons: First, the worst-case runtime of linear program solvers is quite large and exceeds the runtime of our combinatorial algorithm. The second and for our purposes of gaining an understanding of the inherent problem aspects more important reason is that the generality of a linear program solver completely obscures the underlying structure of the problem. Since our model is only a first step towards a more realistic energy-harvesting model (featuring, for example, online aspects, battery capacities, and variable energy rates) it seems desirable to expose and preserve such structural aspects.

Our main result is a polynomial-time combinatorial algorithm when the processor speeds are *well-separated*, meaning that for all speeds s_i we have $\frac{P_{i+1}-P_i}{s_{i+1}-s_i} \geq (1+\epsilon)\frac{P_i-P_{i-1}}{s_i-s_{i-1}}$. To understand this condition, note that there is a strong convex relationship between the speed and the power in CMOS-based processors Most commonly this is modeled assuming the (dynamic) power is the speed cubed. Thus, while higher speeds give better performance, lower speeds give significantly better energy efficiency in terms of energy used per computation step. A chip designer generally wants to choose discrete speeds, from the continuous range of options, that are well separated in terms of performance and energy efficiency. By simple algebra, a sufficient and natural condition for well-separation is that the speeds selected by the chip designer satisfy $s_{i+1} = (1+\delta)s_i$ for some positive δ, and each $P_i = s_i^\alpha$ for some

$\alpha > 1$.

Our algorithm can be viewed as a homotopic optimization algorithm that maintains an energy optimal schedule the recharge rate is continuously decreased. One should note that while our algorithm's runtime (see Theorem 98) is relatively high, this stems largely from simplifying analysis assumptions. A more careful analysis can lower the exponent significantly.

Both our algorithm design and analysis are quite involved and require significant understanding of the evolution of recharge rate optimal schedules. Thus, we start with an informal overview in the next section. The full, formal model description and definitions can be found in Sections 4.2 and 4.3. The actual algorithm description is given in Section 4.4.2.

4.1 TECHNICAL OVERVIEW

The aim of this section is to provide a high level overview of both the design and analysis of our algorithm. We first explain how to recognize an optimal schedule, then give a simple example to illustrate the top level algorithmic approach, then explain how to obtain a continuous algorithm, and finally explain how to obtain a polynomial-time discrete algorithm when speeds are well-separated.

We consider the natural linear program with indicator variables x_{jit} denoting whether job j is run at speed i at time t, and with a variable R denoting the recharge rate. Interpreting the complementary slackness conditions combinatorially, we find that a schedule is optimal if and only if it satisfies the following four conditions:

(a) **Feasibility:** The schedule is feasible. That is, every job is processed to the extent of its size between its release time and deadline and there is always sufficient energy stored in the battery.

(b) **Local Energy Optimality:** The subschedule within each *depletion interval* uses the least possible energy. A depletion interval is the time interval between two consecutive *depletion points* (times when the battery has been entirely depleted).

(c) **Speed Level Relation (SLR):** When two jobs i and j are both run in two depletion intervals I_a and I_b, the difference between job i's *speed levels* in I_a and I_b is the same as the

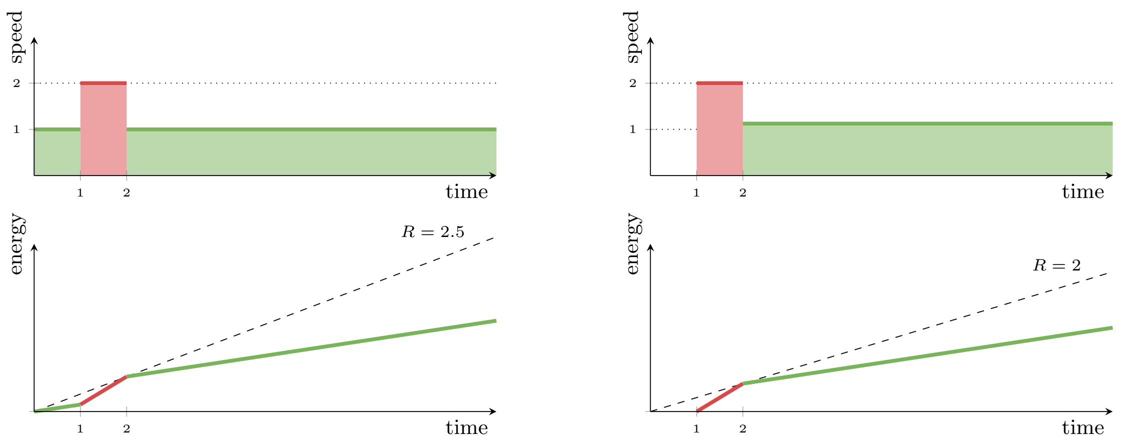


Figure 9: The energy optimal schedule and the recharge rate optimal schedule.

difference between j's speed levels in these intervals. If a job is run at speed $s \in [s_{i-1}, s_i]$ then we say that it is run at speed level i. The relative simplicity of this condition relies on the speeds being well-separated. The fact that the speeds are well-separated is only used in our algorithm design and analysis by application of the SLR condition.

(d) **Saturated Depletion Point (SDP):** There is a depletion point such that any job alive after this point is completely processed after it.

In order to build intuition, let's consider the following simple example instance. The processor can be run at speed 1 with power 1, or at speed 2 with power 4. Job j_1 is released at time 0 with deadline 10 and work 9, and j_2 is released at time 1 with deadline 2 and work 2. The energy optimal schedule would run job j_2 at speed 2 during the time interval $[1, 2]$ and run job j_1 at speed 1 during the time intervals $[0, 1]$ and $[2, 10]$. The minimal recharge rate at which this schedule is feasible is $R = 2.5$. Note that this schedule satisfies the first three optimality conditions, but does not satisfy the SDP condition since j_1 is run both before and after the only depletion point, which is at time 2. One can achieve a smaller recharge rate by moving some of the processing done on j_1 during the time interval $[0, 1]$ to the time interval $[2, 10]$ (see Figure 9).

The high level intuition behind our algorithm is that we start with an energy optimal schedule, and a recharge rate R such that this schedule satisfies the first three optimality conditions. The algorithm then lowers R, while maintaining a schedule satisfying the first three optimality conditions, until the SDP condition is satisfied. As a consequence, the maintained schedule maximizes the energy in the battery at the time of the last deadline subject to the constraint that the schedule is feasible for the current recharge rate.

The algorithm conceptually operates on the *distribution* multigraph G, in which the vertices are the depletion intervals, and there is a directed edge (I_a, I_b) for each distinct way in which work can be transffered from depletion interval I_a to depletion interval I_b. The algorithm tries to find a *transfer path collection*, which is a collection of paths consisting of one path from each depletion interval I_a to the rightmost depletion interval. Given such a transfer path collection, it is relatively straight-forward to determine how to move work to maintain the first three optimality conditions as the recharge rate decreases. There will be a general trend of the work to move later in time, but sometimes it will be necessary to move work earlier in time. There are three types of events that stall progress using this transfer path collection:

- **Edge Removal Event:** It is no longer possible to transfer work on a particular edge because there is no more work left on the corresponding job in the source depletion interval.
- **Speed Level Event:** Further transfer of work using the transfer path collection would cause a job to change its speed level (violating the SLR condition).
- **Depletion Point Appearance Event:** A new depletion point is created.

To make further progress, the algorithm first attempts to find a different transfer path collection. If this is not possible, the algorithm updates the graph G in one of two possible ways:

- **Depletion Point Removal Update:** The algorithm removes a depletion point. The removal of the constraint that the battery must be depleted at this time allows for new ways to transfer work.
- **Cut Update:** The speed levels of some jobs currently running at a discrete speed s_i are updated to an adjacent speed level. This allows for new ways to transfer work without violating the SLR condition. The jobs whose speed has reached one of the processor speeds induce a cut in the graph G. Essentially the dual variables associated with these

jobs are updated so as to both remove the cut and maintain the SLR condition. Although it seems likely, we do not know how to prove that a naive implementation of this process terminates, or converges to the optimal solution. However, we can show that if none of these possible updates creates a multigraph in which there exist a transfer path collection, then the SDP condition holds, and the current recharge rate is optimal.

One can discretize the continuous algorithm by calculating, given the transfer rates for the current transfer path collection, the next edge removal or speed level event, and then discretely transfer enough work to reach that next event.

To obtain convergence to the optimal solution, and a polynomial bound on the number of events (and thus on the total time), we design our algorithm so as to maintain the following hierarchy of monotonicity invariants:

- **Cut Invariant:** Cut updates are at the top of the hierarchy. Intuitively, the cut invariant states that speed level increases monotonically move later in time.
- **Depletion Point Removal Invariant:** Depletion point removal updates occur below cut updates in the hierarchy (and at the same level as speed level events). The depletion point removal invariant is that once a depletion point is removed, it will not be added again until the next cut update. The number of depletion point appearance events can be bounded by the number of depletion point updates.
- **Speed Level Invariant:** Speed level events again occur below cut updates (and at the same level as depletion point removal updates). Intuitively, the speed level invariant is that this event causes the creation of a time interval to which no work is added until the next cut update event.
- **Edge Removal Invariant:** Edge removal events occur at the bottom of the hierarchy. The edge removal invariant is that once work from a job is transferred earlier in time, it will not be transferred later in time until the next cut update, depletion point removal update, or speed level event.

In order to maintain this hierarchy of monotonicity invariants, we must be careful about the way in which two aspects of the algorithm are implemented. The first issue is that G may be exponentially large. We fix this by changing the algorithm so that it will search for a transfer path collection in a subgraph H of G. For every pair I_a and I_b of vertices in G,

H only contains the "best" edge between I_a and I_b. Intuitively, the best edge is the one that involves the most transfers of work later in time. This is sufficient to guarantee that if there is a transfer path collection in G, then there will also be a transfer path collection in H. The second issue is the way that a transfer path collection is selected in G. Intuitively, the algorithm primarily prefers transfers of work later in time, and among such transfers, it secondarily prefers shorter transfers; Conversely among transfers earlier in time, the algorithm secondarily prefers longer transfers.

4.2 STRUCTURAL OPTIMALITY VIA PRIMAL-DUAL ANALYSIS

We model our problem as a linear program and use complimentary slackness conditions to derive structural properties that are sufficient for optimality. These structural properties are used in both the design and the analysis of the algorithm.

We consider the problem of scheduling a set of n jobs $J \coloneqq \{1, 2, \ldots, n\}$ on a single processor that features k different speeds $0 < s_1 < s_2 < \cdots < s_k$ and that is equipped with a solar-powered battery. The battery is attached to a solar cell and recharges at a rate of $R \geq 0$. The power consumption when running at speed s_i is $P_i \geq 0$. That is, while running at speed s_i work is processed at a rate of s_i and the battery is drained at a rate of P_i.

Each job $j \in J$ comes with a *release time* r_j, a *deadline* d_j, and a processing volume (or work) p_j. For each time t, a schedule S must decide which job to process and at what speed. Preemption is allowed, so that a job may be suspended at any point in time and resumed later on. We model a schedule S by two functions $S(t)$ (*speed function*) and $J(t)$ (*scheduling policy*) that map a time $t \in \mathbb{R}$ to a speed index $S(t) \in \{1, 2, \ldots, k\}$ and a job $J(t) \in J$. Jobs can only be processed within their *release-deadline interval* $\big[r_j, d_j\big)$. Thus, a *feasible schedule* must ensure that $J^{-1}(j) \subseteq \big[r_j, d_j\big)$ holds for all jobs j. Moreover, a feasible schedule must finish all jobs and must ensure that the energy level of the battery never falls below zero. More formally, we require $\int_{J^{-1}(j)} s_{S(t)}\,\mathrm{d}t \geq p_j$ for all jobs j and $\int_0^{t_0} P_{S(t)}\,\mathrm{d}t \leq Rt_0$ for all times t_0. Our objective is to find a feasible schedule that requires the minimum recharge rate.

$$\begin{aligned} \min \quad & R \\ \text{s.t.} \quad & \sum_{t \in [r_j, d_j)} \sum_i x_{jit} s_i \geq p_j && \forall j \\ Rt - & \sum_{t' \leq t} \sum_{j \in J} \sum_{i=1}^{k} x_{jit'} P_i \geq 0 && \forall t \\ & \sum_{j \in J} \sum_{i=1}^{k} x_{jit} \leq 1 && \forall t \\ & x_{jij} \in \{0, 1\} && \forall j, i, t \end{aligned}$$

(a) ILP for our scheduling problem.

$$\begin{aligned} \max \quad & \sum_{j \in J} \alpha_j p_j - \sum_t \gamma_t \\ \text{s.t.} \quad & \alpha_j s_i - \sum_{t' \geq t} \beta_{t'} P_i - \gamma_t \leq 0 && \forall j, i, t \\ & \sum_t \beta_t t \leq 1 \\ & \alpha_j, \beta_t, \gamma_t \geq 0 && \forall j, t \end{aligned}$$

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

Figure 10: Primal-dual formulation of our problem.

For the following linear programming formulation, we discretize time into equal length[1] time slots t. Without loss of generality, we assume that their length is such that there is a feasible schedule for the optimal recharge rate R that processes at most one job at at most one discrete speed in each single time step. Our linear program uses indicator variables x_{jit} that state whether a given job j is processed at a speed s_i during time slot t. Note that not only does this imply possibly exponentially many variables but it is not even clear how to choose the length of the time slots. Nevertheless, this will not influence the running time of our algorithm, since we merely use the linear program to extract sufficient structural properties of optimal solutions.

With the variables x_{jit} as defined above and a variable R for the recharge rate, the integer linear program (ILP) shown in Figure 10a corresponds to our scheduling problem. The first set of constraints ensure that each job is finished during its release-deadline interval, while the second set of constraints ensures that the battery's energy level does not fall below zero. The final set of constraints ensures that the processor runs at a constant speed and processes at most one job in each time slot.

[1]By rescaling the problem parameters, we can assume time slots of unit length.

The complementary slackness constraints for the programs shown in Figure 10 give us necessary and sufficient properties for the optimality of a pair of feasible primal and dual solutions.

$$x_{jit} > 0 \quad\Rightarrow\quad \alpha_j s_i - \sum_{t' \geq t} \beta_{t'} P_i - \gamma_t = 0, \tag{4.1}$$

$$R > 0 \quad\Rightarrow\quad \sum_t \beta_t t = 1, \tag{4.2}$$

$$\alpha_j > 0 \quad\Rightarrow\quad \sum_{t \in [r_j, d_j)} \sum_i x_{jit} = p_j, \tag{4.3}$$

$$\beta_t > 0 \quad\Rightarrow\quad \sum_{t' \leq t} \sum_{j \in J} \sum_{i=1}^{k} x_{jit'} P_i = Rt, \tag{4.4}$$

$$\gamma_t > 0 \quad\Rightarrow\quad \sum_{j \in J} \sum_{i=1}^{k} x_{jit} = 1. \tag{4.5}$$

Although these conditions are only necessary and sufficient for optimal solutions of the ILP's *relaxation*, our choice of the time slots' lengths (see above) ensures that there is an integral solution to the relaxation. Based on these complementary slackness constraints, we derive some purely combinatorial structural properties (not based on the linear programming formulation) that will guarantee optimality. To this end, we will consider *speed levels* of jobs in depletion intervals essentially the discrete speed a job reached in a specific depletion interval and how they change at depletion points.

Definition 67 (Speed Level Relation)**.** *A schedule S and a recharge rate R obey the* Speed Level Relation *(SLR) if there exist* speed levels $\mathcal{L}(j, \ell) \in \mathbb{N}$ *with:*

(a) job j processed at speed $s_{j,\ell} \in (s_{i-1}, s_i)$ in depletion interval $I_\ell \Rightarrow \mathcal{L}(j, \ell) = i$

(b) job j processed at speed $s_{j,\ell} = s_i$ in depletion interval $I_\ell \Rightarrow \mathcal{L}(j, \ell) \in \{ i, i+1 \}$

(c) jobs j, j' both active in depletion intervals I_{ℓ_1} and I_{ℓ_2} with $\ell_1 < \ell_2 \Rightarrow \mathcal{L}(j, \ell_2) - \mathcal{L}(j, \ell_1) = \mathcal{L}(j', \ell_2) - \mathcal{L}(j', \ell_1) \in \mathbb{N}_0$

(d) job j processed in $I_l \Rightarrow \mathcal{L}(j, l) \geq \mathcal{L}(j', l)$ for all j' active in $I_{l,j} = I_l \cap [r_j, d_j)$

Using this definition, we are now ready to characterize optimal schedules in terms of the following combinatorial properties. Note for condition (b), a YDS subschedule would be sufficient. We defer the proof to the appendix.

Theorem 68. *Consider a feasible schedule S and a recharge rate R. The following properties are necessary and sufficient for S and R to be optimal:*

(a) S is feasible.

(b) The subschedule within each depletion interval uses the least possible energy.

(c) The SLR holds.

(d) There is a split depletion point: a depletion point τ_k such that no job with deadline greater than τ_k is processed before τ_k.

Proof. The feasibility of S immediately gives us a set of feasible primal variables that fulfill Equations (4.3) and (4.5) of the complementary slackness conditions. Moreover, note that by making the time slots suitably small, the x-variables of the primal solution can be assumed to be integral[2]. In the following, we show how to define a set of feasible dual variables such that the remaining complementary slackness conditions hold. This immediately implies optimality.

Before we define the dual variables, let us define some helper variables that describe how speed levels change at depletion points. Fix a set of speed levels that adheres to the SLR[3] and consider the depletion points $0 < \tau_1 < \tau_2 < \cdots < \tau_L$ of S. By (d), there is at least one depletion point τ_k such that no job j with $d_j > \tau_k$ is processed in $[0, \tau_k)$. Without loss of generality, let τ_k be the leftmost depletion point with this property. By this choice, for any depletion point τ_ℓ with $\ell \in \{ 1, 2, \dots, k-1 \}$ there is a job j_ℓ that is active immediately before and after τ_ℓ. The speed level of j_ℓ increases by $\mathcal{L}(j_\ell, \ell+1) - \mathcal{L}(j_\ell, l) \in \mathbb{N}_0$ from the ℓ-th to the $\ell+1$-th depletion interval. By the SLR, this increase is independent of the concrete choice of j_ℓ (any active job's speed level increases by the same amount from I_ℓ to $I_{\ell+1}$). Thus, we can define $a_\ell \coloneqq \mathcal{L}(j_\ell, \ell+1) - \mathcal{L}(j_\ell, \ell) \in \mathbb{N}_0$ as the increase of speed level at τ_ℓ.

We are now ready to define the dual variables. For $t \notin \{ 1, 2, \dots, k-1 \}$ we set $\beta_t = 0$.

[2]To see this, first note that we can normalize any (also an optimal) schedule using the EDF scheduling policy. By choosing the time slots small enough, each job is processed alone and at a constant speed within a slot. Note that we don't need to know the time slots' size for this argument, the mere existence of such time slots is sufficient (since the resulting optimality conditions are oblivious of the time slots).

[3]If we can choose, we choose the smallest possible speed level.

The remaining β_t variables are defined by the (unique) solution to the following system of linear equations:

$$\begin{aligned} &\sum_{t \geq \tau_\ell} \beta_t = 2^{a_\ell} \sum_{t \geq \tau_{\ell+1}} \beta_t \quad \text{for } \ell \in \{1, 2, \ldots, k-1\}, \\ &\sum_{\ell=1}^{k} \tau_\ell \beta_{t_\ell} = 1. \end{aligned} \tag{4.6}$$

To set α_j, fix a job j and let I_ℓ be the first depletion interval in which j is processed. We set $\alpha_j = \Delta_{\mathcal{L}(j,\ell)} \cdot \sum_{t \geq \tau_\ell} \beta_t$. To set γ_t, remember that we assume time slots to be small enough that at most one job is processed. If no job is processed, we set $\gamma_t = 0$. Otherwise, let j be the job processed in t and set $\gamma_t = \alpha_j s_{\mathcal{L}(j,\ell)} - \sum_{t' \geq t} \beta_{t'} P_{\mathcal{L}(j,\ell)}$.

By construction, these variables fulfill all the remaining complementary slackness conditions (and the second dual constraint). Moreover, it is easy to see that $\alpha_j \geq 0$ for any job j and $\beta_t \geq 0$ for any time slot t. Thus, it remains to show that for all t we have $\gamma_t \geq 0$ and that for all j, i, and t we have $\alpha_j s_i - \sum_{t' \geq t} \beta_{t'} P_i - \gamma_t \leq 0$. For the inequality $\gamma_t \geq 0$, let j be the job processed in t (if there is no job, we have $\gamma_t = 0$ by definition) and let I_l be the depletion interval that contains t. Then the desired inequality follows from

$$\frac{\gamma_t}{s_{\mathcal{L}(j,\ell)} \sum_{t' \geq t} \beta_{t'}} = \Delta_{\mathcal{L}(j,\ell)} - \frac{P_{\mathcal{L}(j,\ell)}}{s_{\mathcal{L}(j,\ell)}} \geq 0 \tag{4.7}$$

(the last inequality follows from the properties of the power function). Now fix a job j, a speed index i, and a time slot t in which j is active. Let l denote the depletion interval that includes t and let j' be the job that is actually processed in t. We have to show $\alpha_j s_i - \sum_{t' \geq t} \beta_{t'} P_i \leq \gamma_t$. This is trivial if $\alpha_j = 0$. Otherwise, using the definition of α_j and γ_t and dividing by $\sum_{t' \geq t} \beta_{t'}$, it is equivalent to

$$\begin{aligned} \iff \quad & \Delta_{\mathcal{L}(j,\ell)} s_i - P_i \leq \Delta_{\mathcal{L}(j',\ell)} s_{\mathcal{L}(j',\ell)} - P_{\mathcal{L}(j',\ell)} \\ \iff \quad & P_{\mathcal{L}(j',\ell)} - P_i \leq \Delta_{\mathcal{L}(j',\ell)} s_{\mathcal{L}(j',\ell)} - \Delta_{\mathcal{L}(j,\ell)} s_i. \end{aligned}$$

Since we assume YDS is used in between depletion points, we know j' runs at least as fast in I_ℓ as j. Thus we have $s_{j',\ell} \geq s_{j,\ell}$ and, in particular, $\mathcal{L}(j',\ell) \geq \mathcal{L}(j,\ell)$. Thus, it is sufficient to show $P_{\mathcal{L}(j',\ell)} - P_i \leq \Delta_{\mathcal{L}(j',\ell)}(s_{\mathcal{L}(j',\ell)} - s_i)$, which follows easily from the properties of the power function. □

4.3 NOTATION

Given a schedule S, we need a few additional notions to describe and analyze our algorithm.

Definition 69 (Power Function Slopes)**.** *For the speeds s_i and their powers P_i we define $\Delta_i := \frac{P_i - P_{i-1}}{s_i - s_{i-1}}$.*

In the simplified model, we assume $\Delta_i = 2^{i-1}\Delta_1$ (the constant 2 is chosen merely for simplicity, you can think of any constant larger 1).

Let us start by formally defining depletion points and depletion intervals. As noted earlier, depletion points represent time points where our algorithm maintains a battery level of zero and partition the time horizon into depletion intervals. Note that these definitions depend on the current state of the algorithm.

Definition 70 (Depletion Point)**.** *Let $E_S(t)$ denote the energy remaining at time t in schedule S. A depletion point τ_i is defined such that $E_S(\tau_i) = 0$ and the algorithm has labeled this as a depletion point. We let $\tau_0 = 0$ and $\tau_{L+1} = \infty$ where L is the number of depletion points.*

Definition 71 (Depletion Interval)**.** *For all $\ell > 0$ we define a depletion interval $I_\ell = [\tau_{\ell-1}, \tau_\ell)$ and the speeds $s_{j,\ell}$ of jobs j during I_ℓ.*

While moving work between depletion intervals, our algorithm uses the jobs' *speed levels* to guide how exactly to move work around.

Definition 72 (Speed Level)**.** *For all j, ℓ with $I_\ell \cap [r_j, d_j) \neq \emptyset$, the* speed level *$\mathcal{L}(j,\ell)$ of j in I_ℓ is such that if j is processed in I_ℓ, then $s_{j,\ell} \in [s_{\mathcal{L}(j,\ell)-1}, s_{\mathcal{L}(j,\ell)}]$.*

The algorithm initializes the speed level for every depletion interval (even those in which j is not run) based on the initial YDS schedule, and throughout its execution assigns speed levels maintaining SLR.

Next, we give a definition of a slightly weaker version of the well known EDF (Earliest Deadline First) scheduling policy. The idea is to maintain a version of EDF between depletion intervals but, locally, allow slight deviations. For example, we forbid situations where for depletion intervals $I_1, \dots, I_4$ some job j_1 is scheduled in I_1 and I_3 and another job in I_2 and I_4. Disallowing such instances forces the subgraph of paths taken by the algorithm to be laminar, which is useful throughout the analysis.

Definition 73 (first-run sequence)**.** *Assume $d_1 < d_2 < d_3 < \cdots < d_n$. To construct the* first-run *sequence $(I_\ell^j)_{j=1}^k$ of a depletion interval I_ℓ let $j_{\ell_1}, \ldots, j_{\ell_k}$ be the k jobs run in I_ℓ ordered by the first time they are run within I_ℓ. Then, $(I_\ell^j) = (\ell_1, \ldots, \ell_k)$. The first-run sequence of a schedule S is the concatenation of all depletion interval first-run sequences from first to last.*

Definition 74 (Weak EDF)**.** *We say that a schedule satisfies* Weak EDF *if the corresponding first-run sequence (S_k) has the following property. For every $j \in [n]$, let f_j and l_j be the first and last appearances of j in (S_k). Then, for all i such that $f_j < i < l_j$, $S_i > j$.*

This next definitions denote to what extent a given schedule adheres to the structural optimality conditions stated in Theorem 68. We distinguish between schedules that (essentially) adhere to the first two optimality conditions and schedules that, additionally, have the third optimality condition (SLR).

Definition 75 (Nice & Perfect)**.** *We say that a schedule S is* nice *if it is feasible, obeys YDS between depletion points, and satisfies Weak EDF. If, additionally, S fulfills the SLR, we call it* perfect.

We now define ϵ-transfers, the building block for our algorithm. Intuitively, they formalize possible ways to move work around between depletion intervals. Our definition will ensure that moving work over ϵ-transfers maintains niceness throughout the algorithm's execution. Moreover, we also ensure that ϵ-transfers only affect the schedule's speed profile at their sources/targets.

Definition 76 (ϵ-transfer)**.** *$(\ell_a, j_a)_{a=0}^s$ is an* ϵ-transfer *if we can, simultaneously for all a, move some positive workload of j_a from ℓ_{a-1} to ℓ_a while maintaining niceness and without any speed in $\ell_1, \ldots, \ell_{s-1}$ changing. ℓ_0 and j_0 (resp, ℓ_s and j_s) are the* source *and* source job *(resp,* destination *and* destination job*) of the ϵ-transfer, respectively. We say that the ϵ-transfer is* active *if it also maintains perfectness.*

Let $T_1 = (\ell_a^1, j_a^1)_{a=0}^{s_1}$ and $T_2 = (\ell_a^2, j_a^2)_{a=0}^{s_2}$ be two different ϵ-transfers with $\ell_0^1 = \ell_0^2$ and $\ell_{s_1}^1 = \ell_{s_2}^2$. Let $a_1^* = \arg\min_a\{\ell_a^1 \neq \ell_a^2\}$ and $a_2^* = \arg\min_a\{j_a^1 \neq j_a^2\}$. We say that T_1 is higher priority *than T_2 if* *(a)* $\ell^2_{a_1^*} < \ell^1_{a_1^*-1} < \ell^1_{a_1^*}$*, or*

(b) $\ell^1_{a_1^*} < \ell^2_{a_1^*}$*, and either* $\ell^2_{a_1^*} < \ell^1_{a_1^*-1}$ *or* $\ell^1_{a_1^*-1} < \ell^1_{a_1^*}$*, or*

(c) a_1^* *does not exist and the deadline for* $j^1_{a_2^*}$ *is earlier than the deadline for* $j^2_{a_2^*}$*.*

Our algorithm compares ϵ-transfers based on source and destination. Once the source and destination have been fixed, the priority comparator is used to determine which ϵ-transfer is used. The following multigraph is used to keep track of legal ϵ-transfers.

Definition 77 (Distribution Graph)**.** *We define* $G_D = (V_D, E_D)$ *as the* distribution graph *where* V_D *is the set of depletion points and for every active* ϵ*-transfer* $(\ell_a, j_a)_{a=0}^s$*, there is a corresponding edge with source* ℓ_0 *and destination* ℓ_s*.*

Finally, the last additional notion we'll be using captures when a job can move work to a given depletion interval. This will be of particular importance when adjusting speed levels, as we have to make sure that these remain consistent.

Definition 78 (Reachable)**.** *A depletion interval* ℓ *is* reachable *(resp.,* actively reachable*) by* j *if there is an* ϵ*-transfer (resp., active* ϵ*-transfer) with source job* j *and destination* ℓ*.*

4.4 ALGORITHM DESCRIPTION

This section provides a formal description of the algorithm. From a high level, the algorithm can be broken into two pieces: (a) choosing which ϵ-transfers to move work along (in order to lower the recharge rate), and (b) handling events that cause any structural changes. We start in Section 4.4.1 by describing the structural changes our algorithm keeps track of and by giving a short explanation of each event. Section 4.4.2 describes our algorithm.

4.4.1 Keeping Track of Structural Changes

How much work is moved along each single ϵ-transfer depends inherently on the structure of the current schedule. Thus, intuitively, an event is any structural change to the distribution graph or the corresponding schedule while we are moving work. At any such event, our algorithm has to update the current schedule and distribution graph. The following are the basic structural changes our algorithm keeps track of:

- **Depletion Point Appearance:** For some job j, the remaining energy $E_S(d_j)$ at j's deadline becomes zero and the rate of change of energy at d_j is strictly negative. If we were not to add this depletion point, the amount of energy available at d_j would become negative, violating the schedule's feasibility. We can easily calculate when this happens by examining the rate of change of R as well as as the rate of change of $s_{j,\ell}$ for all jobs j run in the depletion interval I_ℓ containing d_j.
- **Edge Removal:** An edge removal occurs when, for some job j, the workload of j processed in a depletion interval I_ℓ becomes zero. In other words, all of j's work has been moved out of I_ℓ. Similar to before, we can easily keep track of the time when this occurs for any job j processed in a given depletion interval, since all involved quantities change linearly.
- **Edge Inactive:** An edge inactive event occurs when for some job j its speed $s_{j,\ell}$ in a depletion interval I_ℓ becomes equal to some discrete speed s_i. Once more, we keep track of when this happens for each job processed in a given depletion interval.

Note that by moving work along ϵ-transfers between two events e_1 and e_2, our algorithm causes (a) the speed of exactly one YDS critical interval in each depletion interval to decrease and (b) the speed of some YDS critical intervals to increase. For a single critical interval, these speed changes are monotone over time (between two events). However, critical intervals might merge or separate during this process (e.g., when the the speed of a decreasing interval becomes equal to a neighbouring interval). In other words, the critical intervals of a given depletion interval might be different at events e_1 and e_2. On first glance, this might seem problematic, as a critical interval merge/separation could cause a change in the rate of change of the critical interval's speed, perhaps with the result that the algorithm stops for events spuriously, or misses events it should have stopped for. However, since only neighbouring critical intervals can merge and separate, this can be easily handled: In any depletion interval, there are at most $O(n)$ critical intervals at event e_1. Since only neighbouring critical intervals can merge/separate when going from e_1 to e_2, for each critical interval changing speed there are at most $O(n^2)$ possible candidate critical intervals that can be part of at event e_2. We just compute the next event caused by each of these candidates, and whether or not each candidate event can feasibly occur. Then, the next event to be handled by our algorithm is simply the minimum of all feasible candidates.

This is a relatively inefficient way to handle critical interval changes, but it significantly simplifies the algorithm description. We leave the description of a more efficient way to handle such “critical interval events” for the full version.

When we have identified the next event, we must update the distribution graph and recalculate the rates at which we move work along the ϵ-transfers. Given the definition of the distribution graph, updating the graph is fairly straightforward. However, it might be the case that after updating the graph, there is no longer a path from every depletion interval to the far right depletion interval. This can be seen as a cut in the distribution graph. In these cases, to make progress, we either have to remove a depletion point or adapt the jobs' speed levels; If both of these fixes are not possible, our algorithm has found an optimal solution. A detailed description of this can be found in Section 4.6.2.

4.4.2 Main Algorithm

Now that we have a description of each event type, we can formalize the main algorithm. A formal description of the algorithm can be found in Listing 4.1. We give an informal description of its subroutines CalculateRates, UpdateGraph, and PathFinding below.

UpdateGraph(T, G_D, S): This subroutine takes an event type T, the distribution graph G_D and the current schedule S and performs the required structural changes. It suffices to describe how to build the graph from scratch given a schedule (computing a schedule simply involves computing a YDS schedule between each depletion point). Now the question becomes: Given two depletion points, how do we choose the ϵ-transfer between these two? While perhaps daunting at first, this can be achieved via a depth-first search from the source depletion interval. Whenever the algorithm runs into a depletion interval it has previously visited in the search, it chooses the higher priority ϵ-transfer of the two as defined by the priority relation.

PathFinding(G_D): We define PathFinding(G_D) in Listing 4.2. Note the details of deter-

```
Set $R$ to be recharge rate that ensures YDS schedule, $S$, is feasible
Let $G_D = (V_D, E_D)$ be the corresponding Distribution Graph.
$G'_D$ = PathFinding($G_D$)
$\Delta, \delta_{j,\ell}, T$ = CalculateRates($G'_D, S$)
while True:
    for each job $j$ and depletion interval $\ell$:
        set $s_{j,\ell} = s_{j,\ell} + \Delta \cdot \delta_{j,\ell}$
    set $R = R - \Delta$
    UpdateGraph(T,G_D,S)
    if(there is a fixable cut):
        fix cut with either a depletion point removal or SLR procedure
    if(there is unfixable cut):
        exit
    $G'_D$ = PathFinding($G_D$)
    $\Delta, \delta_{j,\ell}, T$ = CalculateRates($G'_D, S$)
```

Listing 4.1: The algorithm for computing minimum recharge rate schedule.

mining the highest priority edge are omitted but the implementation is rather straightforward. The priority relation for choosing edges is: First choose the shortest right going edge, and otherwise choose the longest left going edge. While this priority relation itself is rather straightforward, it requires a non-trivial amount of work to show that it yields suitable monotonicity properties to bound the runtime (see Section 4.5).

CalculateRates(G'_D, S): This subroutine takes as input the set of paths from the distribution graph G'_D and the current schedule S. It returns for each job j and each depletion interval ℓ, the rate $\delta_{j,\ell}$ at which $s_{j,\ell}$ should change, T, the next event type, and Δ the amount the recharge rate should be decreased. It is straightforward to see the set of paths chosen by the algorithm G'_D can be viewed as a tree with the root being the rightmost depletion interval. Assuming R is decreasing at a rate of 1, and working our way from

```
Let $S = \{v_L\}$, where $v_L$ is rightmost vertex
while exists an edge $e = (v_1, v_2)$ s.t. $v_1 \in S$, $v_2 \in S$ and $e$ is the highest priority such edge:
    add $v_1$ to S
```

Listing 4.2: The PathFinding subroutine.

the leaves to the root, we can calculate $\delta_{j,\ell}$ such that the rate of change of energy at all depletion points remains 0. With these rates, we can use the previously discussed methods to find both T and Δ.

4.5 RUNTIME ANALYSIS

In this section we provide an analysis on the runtime and correctness of our algorithm. We begin with some notes on how the algorithm handles certain cases, We then bound the number of different events that can occur. Finally, we analyze the runtime of the calculations made by our algorithm in between events.

4.5.1 Intricacies of the Algorithm

In this subsection, we describe informally some intricacies of the algorithm that, though not vital to a high-level understanding of the algorithm, are key in its formal analysis.

The definition of ϵ-transfer allows for many counterintuitive ϵ-transfers: For example, ones that take the same job multiple times, or enter the same critical interval multiple times. It is easy for the depth first search that chooses ϵ-transfers to prune such undesirable ϵ-transfers. Here we describe the set of ϵ-transfers pruned, and why.

- **ϵ-transfers that take the same job multiple times, or enter the same critical interval multiple times.** It is easy to see that such ϵ-transfers are in some sense not minimal, and an ϵ-transfer with strictly fewer edges could be obtained.
- **ϵ-transfers violating Weak EDF.** This allows us to say the subgraph of the distribution graph taken by the algorithm has edges that are laminar, and that the algorithm never has reason to take ϵ-transfers that cross each other, as well as that the edges of ϵ-transfers taken by our algorithm are laminar.
- **ϵ-transfers that take a right edge that is completely contained within a previously taken left edge.** Though less intuitive, one can show that such ϵ-transfers can be replaced by a series of left ϵ-transfers, in a manner similar to that used in the proof of

Lemma 83 below. By disallowing such ϵ-transfers, we can say more about the ϵ-transfers taken by the algorithm, making analysis simpler.

- **ϵ-transfers that take work of a job in the opposite direction of previously taken ϵ-transfers.** This allows us to formally prove that any job's workload moves in two phases in between two cut events: first only right and then only left. This insight is key to bounding the number of edge removal events.

As described, the algorithm does not stop for critical interval events. To gain some insight into how this is accomplished, we briefly describe how to calculate the speed level event where a critical interval's speed becomes the upper speed of its current speed level. In a depletion interval, there may be multiple jobs at a speed level, with different releases and deadlines, and so multiple possible ways such a speed level event can occur. However, we know that if this event occurs, it occurs at a critical interval whose borders are releases (or the first time the job can be run in the depletion interval, according to SLR) or deadlines. Thus, for every pair of releases and deadlines, we can compute which jobs must be run in that interval within the depletion interval. If it were the case that the next event is in fact a speed level event caused by this critical interval, we can calculate when it would occur by looking at the rate at which just these jobs are getting work, and calculating when the resulting critical interval speed would become the maximum for its speed level. By considering all possible critical intervals, we can determine which speed level event will actually occur first. For other types of events, similar calculations can be performed.

4.5.2 Bounding the Runtime

Here we show that the number of events that cause the algorithm to recalculate are bounded by a polynomial. The idea is to first bound the number of *cut events*: situations, in which there is a depletion interval without a path of ϵ-transfers to move work to the far right depletion interval (see Section 4.6.2 for details). Then, we show that between any two cut events, there are only a polynomial number of other events. While using such a hierarchical structure to bound events may artificially increase the runtime bound, it is helpful in simplifying the analysis. We conclude this subsection with Theorem 98, which bounds the total runtime.

Lemma 79. *The algorithm fixes a cut in the distribution graph at most kn^2 times.*

Proof. Observe that Property 1 (c) of Theorem 68 can be restated as follows: for each non-degenerate depletion point i (ordered from left to right) there exists a number $\delta_i \in [k] \cup \{0\}$ such that for any job j and any depletion intervals ℓ_1 and ℓ_2, $\ell_1 < \ell_2$, in which j is active, we have that

$$\mathcal{L}(j, \ell_2) - \mathcal{L}(j, \ell_2) = \sum_{i=\ell_1}^{\ell_2 - 1} \delta_i.$$

The algorithm assigns speed levels to jobs in intervals in which they are not active such that this definition is satisfied whenever j is alive.

For any intermediate schedule $\mathcal{S}$ produced by the algorithm, order jobs by increasing deadline, and let $\delta_{j,\mathcal{S}} = \delta_i$ for the job j with smallest index whose deadline is the same as the ith depletion point, and $\delta_{j,\mathcal{S}} = 0$ otherwise. Consider the following potential function:

$$\Phi(\mathcal{S}) = \sum_{j=1}^{n} j \cdot \delta_{j,\mathcal{S}}$$

Recall that speed levels are only modified when a cut in the Transfer Graph is fixed, so this is the only time that the δ_i change. For a fixed cut, let l be the index of the left depletion point defining the cut (if it exists), and r be the index of the right depletion point defining the cut (which always exists). It is straightforward to observe that the algorithm's modification of speed levels to fix a cut increases δ_r by 1 and, if l exists, decreases δ_l by 1. Thus, since there is at most one depletion point per time, Φ increases by at least one every time a cut in the Transfer Graph is fixed (and this is the only event that changes Φ). It is also clear that $0 \leq \Phi \leq kn^2$, since for any i, $\delta_i \leq k$, and the Lemma follows. □

Before we can bound the number of events that occur between cut events, we need several auxiliary results. These form the most technical result of the paper, but turn out to provide strong tools, such that bounding the actual events later on will be relatively straightforward. We first provide some results about our choice of ϵ-transfers. The most important part will be when we introduce *isolated areas*. Intuitively, we will show that during the executing of our algorithm, some time intervals will become isolated in the sense that no workload enters them and any workload that leaves them can do so only in a very restricted way. This turns out a strong monotonic property that helps to bound the number of events.

We say the source (resp., destination) is *outside* an interval $[t_1, t_2]$ if the critical interval the source job (resp., destination job) is running in does not intersect $[t_1, t_2]$, and it is *inside* otherwise.

Observation 80. *When the algorithm chooses an ϵ-transfer T with source s_T and destination d_T, there is a path of ϵ-transfers from the source of that ϵ-transfer to I_L. If T is a left ϵ-transfer, no previously chosen ϵ-transfer has source or destination within $(d_T, s_T]$. If T is a right ϵ-transfer, no previously chosen ϵ-transfer with source or destination within $[s_T, d_T)$ is a left ϵ-transfer.*

Lemma 81. *If two critical intervals C_1 and C_2 in the same depletion interval, with C_1 to the left of C_2, both have active ϵ-transfers to the same destination depletion interval ℓ, then there is an active ϵ-transfer from C_2 with destination ℓ that is higher priority than all active ϵ-transfers from C_1 with destination ℓ.*

Proof. This follows easily from the priority definition of ϵ-transfers in Definition 76. □

Definition 82 (ϵ-transfer span and crossing)**.** *For an ϵ-transfer T, let l be the time that the leftmost of source and destination critical intervals of T begins, and r be the time that the rightmost of source and destination critical intervals of T ends. Then $[l, r]$ is the* span *of T. Two ϵ-transfers T_1 and T_2 are* crossing *if their source and destination critical intervals are all unique, and the intersection of their spans is nonempty.*

Lemma 83. *Let T_1 and T_2 be two ϵ-transfers that cross. Then there is an path of ϵ-transfers T_3 with source that of T_1 and destination that of T_2, and every intermediate destination and source is active. Additionally, if the source of T_1 is decreasable, and the destination of T_2 is increasable, then T_3 is active.*

Proof. We split the proof into cases, based on the directions of T_1 and T_2, and their sources and destinations. Let a_f be the final index of depletion intervals of T_2. We illustrate only one case, as the remaining cases use essentially the same arguments.

Case 1: T_1 is a right ϵ-transfer and T_2 is a right ϵ-transfer, and the source of T_1 is left of the source of T_2. Let a_1 be the lowest index such that ℓ_{a_1} of T_1 is or is to the right of the critical interval from ℓ_{a_2} of T_2, and to the left of the critical interval of ℓ_{a_2+1} of T_2, which must exist since T_1 and T_2 are crossing. Let e be the edge from ℓ_{a_2} to ℓ_{a_2+1} in T_2. If the T_1 critical interval in ℓ_{a_1} is the T_2 critical interval in ℓ_{a_2}, then it is clear that we can create $T_3 = (\ell_a, j_1)_{a=0}^{a_1-1} \cup (\ell_a.j_a)_{a=a_2}^{a_f}$. Otherwise, it must be that $\ell_{a_1} \neq \ell_{a_2}$, since they would have to run in the same critical interval otherwise, as the deadline of j_{a_2} cannot be before $\ell_{a_2} + 1$, and the release time of j_{a_1} cannot be after ℓ_{a_2}. Note also that j_{a_1} must have a later deadline than j_{a_2}, since it can move to ℓ_{a_1} without violating Weak EDF, and similarly, j_{a_2} completes in or before ℓ_{a_1}. Let $T' = (\tilde{\ell}_a, \tilde{j}_a)_{a=0}^{k}$ be the longest path of ϵ-transfers such that each j_a has earlier deadline than j_{a_1}, and each j_a can move work into the critical interval where $j_a - 1$ completes. Either such a T' exists, or j_{a_1} can be run in ℓ_{a_2+1} (in which case we can take T' to be empty). Thus we obtain $T_3 = (\ell_a, j_1)_{a=0}^{a_1-1} \cup (\ell_{a_1-1}, j_{a_1-1}) \cup T'(\ell_a.j_a)_{a=a_2}^{a_f}$ (which is possibly non-minimal, but can be reduced in size). □

Lemma 84. *If at any event, the algorithm chooses ϵ-transfers T_1 and T_2, then T_1 and T_2 are not crossing.*

Proof. This follows as an easy consequence of both Lemma 83 as well as the definition of Weak EDF. □

We now introduce notation and a definition that will be helpful in the coming proofs. Fix any cut event, and let $\Gamma = \{1, \ldots, \tau\}$ denote the events, in order, that the algorithm stops for between that event and the next cut event.

Definition 85 (Isolated Area). *Let $\gamma \in \Gamma$ and t be some time in the schedule, and ℓ_t be the depletion interval containing t. Then the interval $[t, t']$ is an* isolated area*, denoted by $\nu_t(\gamma)$ if it is possible to assign speeds to jobs in ℓ_t such that they obey their releases and deadlines, are run in earliest deadline first order, and t' is a depletion point after t such that there is no active ϵ-transfer with source inside $[t, t']$, and destination outside $[t, t']$, without crossing t in ℓ_t (i.e., any ϵ-transfer with destination in ℓ_t can place work to the right of t in ℓ_t, and any ϵ-transfer with destination in a depletion interval either before ℓ_t or after t' can be split into* *two active halves: one with destination in ℓ_t to the right of t, and the other with source ℓ_t to the left of t). Additionally, the isolated interval is* maximal *if t' is the latest deplation point satisfying this definition. ℓ_t is referred to as the* exit *of $\nu_t(\gamma)$.*

Observation 86. *For any isolated area, the ϵ-transfer T chosen by the algorithm whose source is exit of the isolated area must have destination outside the isolated area. Additionally, the path of ϵ-transfers to I_L from any depletion interval in the isolated area must include T.*

Lemma 87. *For any maximal isolated area $\nu_t(\gamma)$, no ϵ-transfer taken by the algorithm has source outside $\nu_t(\gamma)$ and destination inside $\nu_t(\gamma)$.*

Proof. Consider any active ϵ-transfer T with source s_T outside of $\nu_t(\gamma)$ and destination inside $\nu_t(\gamma)$. We show that the algorithm does not choose T:

- **Case 1: The source is leftmost depletion interval intersecting $\nu_t(\gamma)$.** This follows immediately from Observation 86.
- **Case 2: The source is to the left of the exit of $\nu_t(\gamma)$.** Let T_e be the ϵ-transfer with source the exit of $\nu_t(\gamma)$. By Observations 80 and 86, if the algorithm chose T, it must have chosen T_e first. If T_e is a left ϵ-transfer, this contradicts Obersvation 80. If T_e is a right ϵ-transfer, by Observation 86, this contradicts Lemma 84.
- **Case 3: The source is to the right of $\nu_t(\gamma)$.** We show that either the algorithm does not take T, or the right border of $\nu_t(\gamma)$ could be extended. Let I_r be the rightmost depletion interval that can be reached by a path of active right ϵ-transfers from s_T, and I_l be the first depletion interval to the right of $\nu_t(\gamma)$.
  (a) If $I_r = I_L$, then the algorithm would take a right ϵ-transfer from s_T.
  (b) If there does not exist an active ϵ-transfer with source between I_l and I_r, and destination to the left of the exit of $\nu_t(\gamma)$, then $\nu_t(\gamma)$ could extend to I_r, contradicting the definition of $\nu_t(\gamma)$.
  (c) If there exists an active ϵ-transfer with source between s_T and I_r, and destination to the left of the exit of $\nu_t(\gamma)$, then the longest such ϵ-transfer would be taken before T, and the path of right ϵ-transfers from s_T would be taken rather than T.
  (d) If there exists an active ϵ-transfer with source between I_l and s_T, and destination to the left of the exit of $\nu_t(\gamma)$, then by Lemma 83, there exists an active ϵ-transfer from

s_T to the left of the exit of $\nu_t(\gamma)$, and this ϵ-transfer is higher priority than T, so the algorithm does not take T. □

Lemma 88. *Let $\nu_t(\gamma)$ be an isolated area. If $\nu_t(\gamma)$ exists and is nonempty, then for any $\gamma' \in \Gamma$ with $\gamma' > \gamma$, the maximal isolated area $\nu_t(\gamma')$ exists and $\nu_t(\gamma') \supseteq \nu_t(\gamma)$.*

Proof. We proceed by induction on events. The base case, for event γ, follows by definition. For the inductive step, suppose the lemma holds at event η, and we will show that the lemma continues to hold at event $\eta + 1$. We accomplish this by showing that any depletion interval that is part of $\nu_t(\eta)$ must be part of $\nu_t(\eta + 1)$, and thus $\nu_t(\eta) \subseteq \nu_t(\eta + 1)$. We first consider the movement of work between events, and second consider the effect of the event $\eta + 1$.

Work Movement. Assume work is moved between η and $\eta + 1$. We (conceptually) stop the algorithm just before enough work is moved to cause $\eta + 1$. We show that if there is an active ϵ-transfer with destination outside of $\nu_t(\eta)$ at this time (for ease, we write at $\eta + 1$), then either the destination is part of $\nu_t(\eta + 1)$, or some active ϵ-transfer with source inside $\nu_t(\eta)$ and destination outside $\nu_t(\eta)$ existed at η, contradicting that $\nu_t(\eta)$ is an isolated area. Let $T = (\ell_a, j_a)_{a=0}^{s}$ be an active ϵ-transfer with source in $\nu_t(\eta)$ and destination outside $\nu_t(\eta)$ at $\eta + 1$, and for $a = 0, \ldots, s$ let C_a be the critical interval in ℓ_a used by that edge of the ϵ-transfer.

We show that an ϵ-transfer R with source inside $\nu_t(\eta)$ and destination outside $\nu_t(\eta)$ must exist at η, and then show that an ϵ-transfer R' with the same properties was active at η. We consider transfer edges i (taking work from ℓ_i to $\ell_i + 1$) one at a time beginning with $s - 1$ down to 0. Let C_d be the current destination critical interval for the ϵ-transfer we are building, which is initially C_s. We show that, assuming there is an ϵ-transfer from C_{i+1} to C_d, then there is one from C_i to either C_d or some other critical interval outside $\nu_t(\eta)$ (i.e., we choose a new C_d).

First note that if j_i was present in C_i at η, edge i must exist at η as well (since no event happened), and thus an ϵ-transfer from C_i to C_d exists. Otherwise, some ϵ-transfer T' was taken at η must have moved job j into C_i. If the source of T' is in $\nu_t(\eta)$, then this source has an ϵ-transfer to C_{i+1}, and therefore an ϵ-transfer to C_d, thus we have found R. If the source of T' is outside $\nu_t(\eta)$, then the destination of T' is also outside $\nu_t(\eta)$ by Lemma 87. Thus there is an ϵ-transfer from C_i to the destination of T', so we change C_d to be the destination critical interval of T'.

It remains to construct an ϵ-transfer R' that is active at η. Let s_R and d_R be the source and destination critical intervals of R. We first construct an ϵ-transfer R_1 with destination d_R with a decreasible source at η. If s_R is decreasible at η, then $R_1 = R$. Otherwise, we know s_R is decreasible at $\eta + 1$, so if s_R was not decreasible at η, the algorithm must have taken some ϵ-transfer R_1' with destination s_R at η, and the source of R_1' must be in $\nu_t(\eta)$ by definition of isolated area. Combine R_1' and R to yield R_1, an ϵ-transfer whose source is decreasible. Similarly, if d_R is decreasible at η, then $R' = R_1$. Otherwise, we know d_R is increasible at $\eta + 1$, so if d_R was not increasible at η, the algorithm must have taken some ϵ-transfer R_2' with source d_R at η, and the destination of R_2' must be in ourside of $\nu_t(\eta)$ by Lemma 87. Thus, combine R_1 and R_2' to obtain R'

Events. We show that the event $\eta + 1$ cannot cause the isolated interval to decrease in size or cease to exist. In each case, we show no new active ϵ-transfers with source inside $\nu_t(\eta)$ and destination outside $\nu_t(\eta)$ could become available.

- **Depletion Point Addition Events:** For any depletion point added, the only ϵ-transfers affected are those with source or destination in the depletion interval that gained the depletion point, and whether or not those ϵ-transfers were active did not change.
- **Depletion Point Removal Events:** This even has a similar effect on ϵ-transfers as depletion point addition events, except when the depletion point removed is the rightmost depletion point of $\nu_t(\eta)$. In this latter case, let ℓ_R be the depletion interval that merged with the rightmost depletion interval of $\nu_t(\eta)$. The fact that the depletion point was removed means that there are now no active ϵ-transfers from ℓ_R to any other depletion interval, thus at $\eta + 1$ the isolated area can be expanded to include the (now removed) depletion interval ℓ_R.
- **Edge Inactive Events:** These cause ϵ-transfers to cease to be active, thus no new ϵ-transfers can appear as a result of these events.
- **Critical Interval Merge and Separation Events:** Critical intervals merging and separating can combine or split ϵ-transfers, but do not cause ϵ-transfers to become active from inactive. Thus, the only place where merge events can cause a new active ϵ-transfer

is at the exit of the isolated area; However, these new ϵ-transfers must cross t and thus do not cause the isolated area to cease to exist.

- **Edge Removal Events:** These can only remove destinations for ϵ-transfers, and thus can only enlarge the isolated area. □

With these technical Lemmas we are now ready to bound the number of non cut events. The hierarchy used assumes depletion point additions/removals and speed level events occur at the same level, but below cuts, and that edge removals occur at the bottom of the hierarchy.

Lemma 89. *There are at most $O(n)$ depletion point addition and removal events.*

Proof. We show that, between cut events, once a depletion point is removed, it never returns. Since there are at most n depletion points in the schedule, there can be at most $2n$ depletion point addition or removal events. Intuitively, the removal of a depletion point creates an isolated area, which by Lemma 88 persists. We then argue that, since work is never removed from the right of the old depletion point, no new depletion point can appear there.

Suppose at event γ a depletion point is removed at t, and let t' be the time of the next depletion point. We show that $[t, t']$ is an isolated area. This follows because we do not remove a depletion point unless there is no active ϵ-transfer from the corresponding depletion interval to outside of it.

Fix any depletion interval, and observe that, as work is moved by the algorithm, the total energy available at any time point to the right of the critical interval being decreased must be increasing, due to the fact that the recharge rate is decreasing and the next depletion point must be maintained. Since for any $\gamma' > \gamma$, by Lemma 88 $\nu_t(\gamma')$ exists, and by the fact that the algorithm does not merge critical intervals that appear on both sides of t, no critical interval to the right of t is ever the source of an ϵ-transfer, and thus the energy at t is always increasing, and so t can never be a depletion point again. □

Definition 90 (work barrier)**.** *For a time t, a depletion point t' is a t* work barrier *if there is no active ϵ-transfer with source in $[t, t']$ and destination to the right of t'.*

Lemma 91. *If t' is a t work barrier at γ caused by a right ϵ-transfer as described in Lemma 90, then for any $\gamma' > \gamma$, there is some $t_2 \geq t'$ such that t_2 is a t work barrier at γ'.*

Proof. It is easy to see that for any isolated area $\nu_t(\gamma)$, the right endpoint is a t work barrier. We show that the isolated area $\nu_t(\gamma)$ exists, and thus the work barrier exists at γ' as the right endpoint of $\nu_t(\gamma')$.

Assume the work barrier iss caused by a right ϵ-transfer T taken by the algorithm, then suppose to obtain a contradiction that $[t, t']$ is not an isolated area. Then there is some active ϵ-transfer T' in $[t, t']$ with destination to the left of t, which is the right endpoint of the source of T. By Lemma 83, we can create an active ϵ-transfer from the source of T' to the destination of T, contradicting there is a work barrier at t'. □

Lemma 92. *There are at most $O(n)$ speed level events.*

Proof. Let γ be a lower speed level event, and t be the left border of the critical interval causing this event. We argue that there is an isolated area $\nu_t(\gamma)$. As a result, no job that could be placed to the right of t will ever be the source of an ϵ-transfer again, and thus this critical interval will never cause itself to decrease again (which could cause it to hit a lower speed level again, or cause it to not be at an upper speed level). Since there are at most $O(n)$ critical intervals, there are at most $O(n)$ such events.

We now show that $\nu_t(\gamma)$ exists.

- **Case 1: Lower Speed Level Events.** Consider the ϵ-transfer T, with source C_s that was taken from the critical interval causing the lower speed level event at $\gamma - 1$. Let t be the left endpoint of C_s. There are two cases, depending on the direction of T.
  - **Subcase 1: *T* is a right *ϵ*-transfer.** By Lemma 90, if t_1 is the right endpoint of C_s, there was a t_1 work barrier at some depletion point t', and thus there was no active ϵ-transfer from the right of C_s to the right of t'. Since C_s hit a lower speed level, and no other event occurred, at γ there is no active ϵ-transfer from the start of C_s to the right of t'. If we can show that there is no active ϵ-transfer from $[t, t']$ to the left of t, then we have shown that $\nu_t(\gamma)$ exists. If such an active ϵ-transfer did exist, then it would have crossed T. By Lemma 83 and Lemma 81, we would could construct a higher priority active ϵ-transfer to the destination of T than T, contradicting that the algorithm took T.
  - **Subcase 2: *T* is a left *ϵ*-transfer.** Let t' be the work barrier caused by the

parent of T (T must have a parent, since it is left-going and there is a path from the destination of T to ℓ_L). By the definition of work barrier, there is no active ϵ-transfer in $[t, t']$ with destination to the right of t'. If there were an active ϵ-transfer between t and t' with destination to the left of t, then by Lemma 83 we could construct T' with source to the right of the source of T, and the same destination as T, and thus T' would have been higher priority than T by Lemma 81, so the algorithm would have taken it instead of, or as the parent of, T. Thus $[t, t']$ is an isolated area.

- **Case 2: Upper Speed Level Events.** Consider the longest ϵ-transfer T, with source C_s and destination C_d where C_d is the critical interval causing the upper speed level event at $\gamma - 1$. There are two cases, depending on the direction of T.
  - **Subcase 1: $\boldsymbol{T}$ is a right $\boldsymbol{\epsilon}$-transfer.** Consder the first event that causes C_d to decrease again, and let T' be the ϵ-transfer with source C_d. At this event, if T' is a right ϵ-transfer, let t be the right endpoint of C_d. Then there is a t work barrier. If T' is a left ϵ-transfer, then if t is the depletion point immediately to the right of C_d, there is a t work barrier to the right of C_s. In both cases, by Lemma 91, this work barrier persists, and thus any right ϵ-transfer with destination C_d must be going to the sink of a left ϵ-transfer, contradicting that the algorithm would have chosen it.
  - **Subcase 2: $\boldsymbol{T}$ is a left $\boldsymbol{\epsilon}$-transfer.** Let t be the left endpoint of C_d. Let t' be the work barrier caused by the parent of T (T must have a parent, since it is left-going and there is a path from the destination of T to ℓ_L). By the definition of work barrier, there is no active ϵ-transfer in $[t, t']$ with destination to the right of t'. If there were an active ϵ-transfer between t and t' with destination to the left of t, it can be composed with T by Lemma 83 to obtain a higher priority ϵ-transfer than T that the algorithm could have chosen. Thus $[t, t']$ is an isolated area. □

Definition 93 ($\boldsymbol{(j, \ell)}$ (active) work barrier**).** *Let t_1 be the first time that j can be run in ℓ (according to the speed levels of jobs in ℓ). A time (depletion point) t' is a (j, ℓ)* work barrier *((j, ℓ)* active work barrier*) if no job j' with release time after t_1, and deadline before that of j, is part of an ϵ-transfer (path of active right ϵ-transfers) crossing t' that does not contain an edge taking j, or some other job with earlier deadline than j and release time before t_1,* *from ℓ.*

Lemma 94. *Suppose the algorithm chooses an ϵ-transfer T that contains an edge moving j from ℓ_1 to ℓ_2. Then there is a (j, ℓ_1) (active) work barrier somewhere to the right of ℓ_1 and to the left of where j is run in ℓ_2 (to the right of ℓ_1).*

Proof. We prove the existence of the (j, ℓ_1) work barrier first. For the sake of contradiction, suppose this does not hold, i.e., there exists ϵ-transfer T' that takes some job j' with release time after the first time j can run in ℓ_1, and with deadline before j, that does not contain an edge taking j, or some other job appropriate job, from ℓ_1. Then by Lemma 83, we can compose the pieces of T' and T together to get an ϵ-transfer R taking j' (eventually) to ℓ_2, to the destination of T. We will show that R is active and higher priority than T, contradicting that the algorithm chose T. First note that the edge of T' taking j' must be right going, as the release of j' is within ℓ_1. Additionally, j' must be at the same speed level as j, or else it would not be able to cross the first time j can run in ℓ_2.

If j and j' are in the same critical interval, then we can create an ϵ-transfer from the source of T to the destination of T taking j' instead of j at ℓ_1, which is clearly higher priority than T, as j' is released later and has deadline earlier than j. Otherwise, j' is not at a lower speed level in ℓ_1 since it must be at a higher speed than j, so we can take an ϵ-transfer with j' as the source, which is higher priority than T as long as the source of T is ℓ_1 or to the left of ℓ_1. If the source of T is to the right of ℓ_1, then it must be before ℓ_2, since right edges cannot be below left edges of ϵ-transfers. Note that the edge from T into ℓ_1 before taking j must had source ℓ' to the right of the destination of the edge taking j', or else there would be a Weak EDF violation. However, T' must cross ℓ', since the destination of the j edge in T is to the right of ℓ'. Thus, by Lemma 83, there is a way to compose T and T' to create an ϵ-transfer that does not use j or some other job with earlier deadline than j and release time before the first run time of j in ℓ_1.

We additionally note that there must be a (j, ℓ_1) active work barrier to the right of ℓ_1. Otherwise, we could use the active ϵ-transfer from such a j' as part of a path of ϵ-transfers, which would be higher priority than T. □

Lemma 95. *If t' is a (j, ℓ) (active) work barrier at γ created from j moving work right from ℓ, and there is still some work of j to the left of ℓ, then for any $\gamma' > \gamma$ before j begins moving work left, the algorithm takes no ϵ-transfer crossing t' at γ'.*

Proof. We prove the lemma for work barriers first. Let C_l be the last critical interval that is reachable by some job j' with release after j can be first run in ℓ, and deadline before j. It is clear that j' must be at the same speed level as j for this to be a problem. There are two cases: either the work barrier could be removed by C_l merging with another critical interval, or work from a job from beyond the work barrier enters C_l.

- **C_l merges with another critical interval.** Let T be the ϵ-transfer at γ causing the work barrier. We first show that, at γ, if C_l can merge with another critical interval C_r that would give j' an ϵ-transfer over the work barrier to the destination of j in the ϵ-transfer causing the work barrier, then either C_l is at an upper speed level, or C_r is at a lower speed level. If not, C_r could be the source of an ϵ-transfer with destination that of T, and source to the left of T, making which would be higher priority than T. Additionally, we can obtain a series of ϵ-transfers that would be higher priority than T, depending on two cases
  - **Case 1: j' is merged with j in ℓ.** In this case, T could use j' instead of j and end in C_l.
  - **Case 2: j' is not merged with j in ℓ.** Then j is not at an upper speed level in ℓ, and j' is not at a lower speed level in ℓ. Thus, an active ϵ-transfer exists taking j' to C_l, and T could end in ℓ.

  If C_l decreases due to j' leaving over an edge e, then there is a j'-isolated area at the destination of e, and all ϵ-transfers must go through this depletion interval, and thus an ϵ-transfer crossing the work barrier would have to leave out some point other than the destination of e, contradicting that the algorithm took that ϵ-transfer (see Lemma 97). Thus it must be that C_l is not at an upper speed level, and is not decreasing, and the critical interval it merges with is increasing. The only remaining possibility is that C_r was at a lower speed level at γ. Note that any job $\tilde{j}$ in C_r that could be used when C_r and C_l merge must have deadline after that of j, or be released after ℓ, as otherwise the

ϵ-transfer uses a job with deadline before that of j that was alive in ℓ when j could be first run there, or it contradicts the location of the work barrier at γ. Note also that C_r increasing cannot be due to the addition from the left of j or any other job with release time before the first time j can run in ℓ and deadline before that of j, as this would imply that j' could be taken into C_l instead, contradicting the ϵ-transfer taking the other job was used. Similarly, C_r increasing cannot be due to the addition from the right of j or any other job with release time before the first time j can run in ℓ and deadline before that of j, as this would create a j-isolated area, and no ϵ-transfer would enter it from outside (see Lemma 97).

If C_r increasing is due to some job $\tilde{j}$ with deadline before that of j, it must be coming from the right, as otherwise there would be an active ϵ-transfer from j' using this job already. The source of this ϵ-transfer must be to the left of ℓ, or to the right of C_l. In the first case, we could construct a higher priority ϵ-transfer with same source and destination ending in C_l. In the second case, there is an isolated area at the right endpoint of C_l, and thus the only way to move work out of the isolated area is through C_l, so no ϵ-transfer taken would cross the work barrier.

Now suppose C_r increasing is due to some job $\tilde{j}$ with deadline after that of j. First assume that this ϵ-transfer taking $\tilde{j}$ is a right ϵ-transfer. Note that C_r must be in the destination of j in T. However, by the fact that C_r was not increasing at γ, the destination of j was not the destination of T, implying there is some work barrier to the right of this destination. However, by Lemma 91, this work barrier could not have disappeared, contradicting that a right ϵ-transfer was being taken to this destination of j. On the other hand, this ϵ-transfer is a left ϵ-transfer, there is an isolated area at the right endpoint of C_l, and thus the only way to move work out of the isolated area is through C_l, so no ϵ-transfer taken would cross the work barrier.

- **Work from some job beyond the work barrier enters C_l.** Let this job be $\tilde{j}$. First note that $\tilde{j}$ must be entering C_l from the right, since it must be a higher priority job than j as it's running between two times when j is run, and must be released before the first time j can run in ℓ, or else it would not be able to go beyond the work barrier, but if so then the work barrier definition is not concerned with ϵ-transfers involving such jobs.

Thus, because the edge taking $\tilde{j}$ is a left edge, there is a $\tilde{j}$-isolated I area starting at C_l. Any ϵ-transfer using j' before $\tilde{j}$ would contradict the property that no ϵ-transfers enter the isolated area (see Lemma 97), and thus the algorithm never takes such an ϵ-transfer.

The active work barrier persists via an argument identical to that of Lemma 91, as we can again show there is an isolated area that ends at the work barrier. □

Finally, we bound the number of edge removal events. We will need one Observation regarding the algorithm's avoidance of cycles.

Observation 96. *The algorithm will never take an $\epsilon-$trasfer $(l_a, j_a)_{a=0}^{s}$ such that for $a_1 \neq a_2$, $l_{a_1} = l_{a_2}$. Intuitively this tells us the algorithm will never use an ϵ-transfer with a cycle.*

With this, we now get the following bound on the number of edge removals.

Lemma 97. *The number of edge removal events between cuts is at most $O(n^3)$.*

Proof. The high level idea of the proof is to show that for each job there are two phases of the algorithm between cuts. The first phase involves moving work from this job left to right and the second phase involves moving work right to left. To show this, we demonstrate that whenever a job moves work from left to right there is a work barrier that persists over time. With this work barrier, it can be seen that this job will never move work to the right again. With this in hand, we can show that the number of edge removals for each phase is polynomially bounded. We now formalize this below.

Let j be an arbitrary job and consider the first time there is an $\epsilon-$transfer $T = (l_a, j_a)_{a=0}^{s}$ chosen by the algorithm such that for some a, $j_{a'} = j$ and $l_{a'-1} > l_{a'}$. That is work from j is moved right to left. Let t_1 be the time that j is run in $l_{a'}$. We say that $A_j = [t_1, t_2]$ is a *j-isolated area* if t_2 is the minimum depletion point such that for every job j' run after the first depletion interval in A, say I_{A_j}, either $d_{j'} \leq t_2$ and $r_{j'} \geq r_j$, or everything reachable by j' is currently at an upper speed level. Equivalently, this says that for any job inside the critical interval, the only way of moving work out is through the leftmost depletion interval.

The first step is to show that initially such a *j-isolated area* exists. Assume by contradiction there is some ϵ-transfer that leaves I_{A_j} through a depletion interval that is not the leftmost depletion interval. There are two cases to consider. Whether the edge leaving is a left going edge or a right going edge.

- **Case: Left going edge** There are two sub cases, depending on whether the source is inside the left edge taken by j or whether the source is outside. In both cases, we argue that you can form a higher priority ϵ transfer.

  Assume there is an active ϵ-transfer T' such that the source of T' is under the left edge chosen by j, that is, $I_{a'} \leq I_{T'_s} \leq I_{a'-1}$, and the destination $I_{T'_d}$ is to the left of t_1 and further T' does not leave I_{A_j} through the leftmost depletion interval. We need to argue that you can combine part of T' with part of the original ϵ-transfer T to get a new ϵ-transfer T'' that is longer than the one chosen, contradicting our choice of T as the longest left-going ϵ-transfer. Specifically, take the original ϵ-transfer until we get to the edge that T uses to leave I_{A_j} and take this instead. By Lemma 83 we can combine these to form T'' To argue that the algorithm would have chosen T'' instead of T all that remains is to argue that the destination of T'' was a sink at the time T was chosen. This is a direct consequence of Lemma 84, that the algorithm does not choose crossing edges. In the second sub case we can use similar approach here, the only difference being that we may need to combine several new ϵ-transfers to reach the same contradiction.

- **Case: Right going edge** We essentially just need to prove the lemma that it is not true that for every depletion point to the right that we can move work over that depletion point with a right going ϵ-transfer. Equivalently, there exists at least one depletion point to the right of the source of j that has no active right going ϵ-transfers over it. If there were no such depletion point then combining this with Lemma 84 would give us a sequence of right going ϵ-transfers that can be connected to the right-most depletion interval. Again this would contradict our choice of T as the algorithm preferences right going before left going ϵ-transfers.

The next step is to show that the properties of I_{A_j} persist over time. Namely, we need to show that the only way to exit I_{A_j} is through the left most depletion interval. There are two things we need to verify. First, that no new work is placed in I_{A_j}, and second, that any critical interval reachable from a job in I_{A_j} remains at an upper speed level. To show the first claim, we consider two cases. In the first case, assume that there is some ϵ-transfer chosen by the algorithm with source outside of I_{A_j} and destination inside I_{A_j}. This implies that when this ϵ-transfer is chosen, there is a path from the destination to the rightmost depletion interval. However this results in either a crossing edge or a cycle, both of which the algorithm forbids.

In the second case assume there is some ϵ-transfer that puts work of some job j' into I_{A_j} but neither the source nor the destination are contained in I_{A_j}. There are two sub cases to consider. If the critical interval j' is being placed into is at a lower speed level, then note that since middle pieces of ϵ-transfers do not change speeds this will not violate any property of I_{A_j}. In the case where j' is not at a lower speed level, this implies that taking the portion of the ϵ-transfer that starts in this critical interval and then leaves I_{A_j} is an active ϵ-transfer, contradicting that all critical intervals currently reachable from I_{A_j} are at upper speed levels.

Lastly, we show that any critical interval reachable from a job in I_{A_j} remains at an upper speed level. Assume by contradiction that some the algorithm chooses some ϵ-transfer with a job j' as the source such that j' is part of a reachable critical interval from I_{A_j} (Indeed this is the only way for a critical interval at an upper speed level to decrease). Note that the destination of this ϵ-transfer is not an an upper speed level. However then by Lemma 83 we can combine this ϵ-transfer with an ϵ-transfer emanating from I_{A_j} contradicting that all critical intervals from I_{A_j} are currently at upper speed levels.

The last step is to show that as a result of the existence of the j-isolated interval, j can no longer move work to the right. Indeed assume by contradiction that at some future time point, j is involved in an epsilon transfer $(l_a, j_a)_{a=0}^{s}$ where $j's$ work is moved from l_{a-1} to l_a for some a such that $l_{a-1} < l_a$. There are three cases to consider.

First consider the case when $l_{a-1} < t_1$ and $l_a > t_1$. Note by the definition of the A, the only way to move work out of A is using the left most depletion interval I_{A_j}. Further, by Weak EDF, there can be no edge with source greater than l_{a-1} and destination greater than l_a or less than l_{a-1}. These together tell us that at some point the algorithm chooses another $\epsilon-$transfer with source l_{a-1}, contradicting Observation 96.

In the second case assume both l_{a-1} and l_a are not contained in A. By definition of an active $\epsilon-$transfer we know that j_s is not at an upper speed level in l_s. However we also know that j_s is reachable by all pieces of j, at least one of which is contained in A_j. This contradicts the second property of A_j, namely all jobs reachable from inside A_j are at upper speed levels.

For the last case, assume that both l_{a-1} and l_a are contained in A. Similar to the first case, since we can only move work out of A from the leftmost depletion interval, at some point the algorithm must choose another ϵ-transfer emanating from l_{a-1} again contradicting Observation 96.

Now that we have established a left and right phase for job j, we can show that there are only polynomially man edge removals for job j.

Suppose a job j is moving out of a depletion interval along a right edge of an ϵ-transfer, and the work of j is completely removed from the depletion interval. Then j never reenters the depletion interval by a right edge of an ϵ-transfer.

We show that any edge that could return the work of j to the depletion interval ℓ must end to the left of a work barrier, which will contradict that the algorithm chose such an ϵ-transfer.

Let γ be the event when the work of j is completely removed from the depletion interval, and T be the ϵ-transfer that removed j. Let ℓ_2 be the destination of j in T. Suppose that at some later event, the algorithm takes another ϵ-transfer T' that moves work from j back into ℓ from the left of ℓ. Our goal is to show that the path of ϵ-transfers from the destination of T' must cross a work barrier, and any such ϵ-transfer that can do this would cause a Weak EDF violation.

We first show that T' must be a right ϵ-transfer. To see this, first note that the edge e taking j in T' is right, so it cannot come after a left edge containing it. Additionally, e cannot come after a series of smaller left edges starting from the right of ℓ, as j must be runnable in the entirety of ℓ, so whatever edge of T' that went left past ℓ must have been able to stop in ℓ, contradicting the algorithm took T'. A left edge going from the right of ℓ to the left of the right endpoint of e is not possible, since, by Lemma 95, from the event when T was taken, there is a (j, ℓ) work barrier somewhere before ℓ_2, so any job taken would have to have deadline before j and been released before j can be first run in ℓ, causing a weak EDF violation if this job were taken and j were moved into ℓ. A series of left edges going from ℓ to the left of ℓ_2 is also not possible, as either the edge crossing ℓ_2 would cause a weak EDF violation, or the algorithm could have taken a more minimal ϵ-transfer not including j, contradicting that it took T'.

By Lemma 94, any destination of T' must be to the left of a (j, ℓ) active work barrier, which persisted by Lemma 95 from the one from the j edge in T, as any job that could cross the work barrier would have to have deadline before j and been released before j can be first run in ℓ, causing a weak EDF violation if this job were taken and j were moved into ℓ. Thus, T' must go to a sink that is the source of a shorter left ϵ-transfer, a contradiction that T' was chosen, or the path of ϵ-transfers from T' must use a job that could cross the work barrier, and thus has to have deadline before j and been released before j can be first run in ℓ, causing a weak EDF violation by j being moved into ℓ, contradicting the algorithm took T' □

Combining all of these, we now give a Theorem bounding the total runtime of our algorithm. Again note that this could likely be improved by removing the hierarchy at the expense of further complicating the analysis.

Theorem 98. *The runtime of Algorithm 4.1 is* $O(n^7k^2)$.

Proof. We first calculate an upper bound on the total times the algorithm stops for some event. Multiplying this by the time spent in between events will give us our final bound. Note that from Lemma 79 there at most $O(kn^2)$ cut events. Between each cut event there at most $O(n)$ depletion point events and $O(n)$ speed level events. Finally, there are at most $O(n^3)$ edge removal events between any two other events. Combining this gives us that line 5 of Algorithm 4.1 will be executed at most $O(kn^6)$ times.

To complete the analysis we need to bound lines $6-8$, line 9, line 11, and line 14. Consider first lines $6 - 8$. Note there are at most $O(n^2)$ unique job depletion interval pairs, and line 7 is done in time $O(1)$. Line 8 also takes time $O(1)$ giving us total time $O(n^2)$.

To bound line 9, the UpdateGraph procedure, recall that we recalculate both the schedule and the graph. To calculate the schedule note there are n depletion intervals each with at most n jobs. Since the YDS algorithm runs in time $O(n \log^2 n)$ our total time to compute the schedule is $O(n^2 \log^2 n)$. For computing the subgraph of the distribution graph recall that for every depletion interval, our algorithm will use a depth first search on n vertices and possible n^2 edges giving a total runtime of $nO(|E|) = O(n^3)$.

To bound line 11, note that checking each depletion point for possible removal takes time $O(n)$ and fixing speed levels takes time at most $O(n^2)$ as there are at most n different speed levels for each job.

Finally to bound line 14 note to calculate the rates such that depletion points remain can be done in $O(n^2)$ since there at most $O(n)$ atomic intervals inside each depletion interval. To calculate the next event, for edge removals takes time $O(n^2)$. For speed level events, since we must consider the possibility that jobs merge along the way there are $O(n^2)$ calculations for each depletion interval, and therefore takes a total of $O(n^3)$.

Combining this, we see that the runtime is $O(kn^6(n^2 \log^2 n + n^3 + n^2 + n^3)) = O(kn^9)$. $\square$

4.6 ALGORITHM CORRECTNESS

In this section we demonstrate our algorithm correctly finds the optimal schedule. At a high level, in we show that at all steps of the algorithm we maintain optimality conditions 1-3 and that when the algorithm can no longer make progress it satisfies the last optimality condition. We first consider movement of work by the algorithm, and its handling of speed level and edge removal events. We then consider cut events.

4.6.1 Non-Cut Events

We now show that moving work and handling non-cut events does not violate any of the optimality conditions maintained throughout the algorithm's execution.

Theorem 99. *The algorithm maintains Properties (a) to (c) of Theorem 68 when moving work and handling non-cut events.*

Proof. Clearly, when the recharge rate is initially set from the YDS schedule of the instance, Properties (a) and (b) are maintained. Property (c) follows from the YDS property of the schedule: First, every job is assigned the single speed level corresponding to the speeds at which it is run when it is run; Second, if there were a time at which a job j is alive but some other job j' was being run, with the speed level of j' less than the speed level of j, this would contradict the YDS property as j' would have to be part of the critical interval of j and thus have the same speed level.

Moving Work. Since the step of moving work does not change the speed levels of jobs, and the process stops when critical intervals hit a new discrete speed (i.e., an edge becomes inactive) or a job no longer has work to move, moving work cannot violate Property (c). Additionally, work is moved in a work-preserving manner for each job, and the process stops if at some new time the total amount of energy available becomes zero (i.e., a depletion point appears), so Property (a) is maintained. Property (b) is maintained by the fact that we have a YDS schedule between critical intervals. By the fact that at non-cut events, only new ϵ-transfers are chosen, the handling of these events does not violate Properties (a) to (c). □

4.6.2 Maintaining the SLR & Reaching Optimality

Consider a situation when there is a depletion interval ℓ for which there is no path to $L+1$ in the distribution graph. This means we are unable to move workload out of this depletion interval and, thus, cannot lower the recharge rate without violating the SLR or other optimality conditions. The following lemmas take a closer look at such situations. In particular, we show that we either can fix the speed levels, adapt the set of depletion intervals, or have found an optimal solution.

We start with the most intuitive reason for not being able to make progress: there are jobs that could still transfer work to the rightmost depletion interval (possibly taking several ϵ-transfers), but the SLR requirement renders any such path inactive. For a single ϵ-transfer, we can easily change the speed levels such that using this ϵ-transfer will not violate Properties (a) or (b) of the SLR. However, to handle (c) and (d), we have to take care to adapt the speed levels of certain jobs in a compatible way. The following lemma takes care of that.

Lemma 100. *Assume there is a depletion interval ℓ_0 without a path of active ϵ-transfers to $L+1$. Furthermore assume there is a path to $L+1$ utilizing at least one inactive ϵ-transfer. Then we can fix the speed levels and increase the set of nodes reachable from ℓ_0.*

Proof. Let $\ell_{\min}$ and $\ell_{\max}$ denote the minimal and maximal depletion intervals reachable from ℓ_0

via a path of active ϵ-transfers. Obviously, no depletion interval in $\{ \ell_{\min}, \ell_{\min}+1, \dots, \ell_{\max} \}$ can reach $L+1$ via a path of active ϵ-transfers. In fact, by definition, no active ϵ-transfer leaves the union of these depletion intervals. Now fix an arbitrary (inactive) ϵ-transfer T leaving $U := \bigcup_{\ell=\ell_{\min}}^{\ell_{\min}} I_l$. Let j denote the job used to move work out of U in T and define the job set J_j of jobs that are part of some (active or inactive) ϵ-transfer that also uses j. Note that either (a) any job $j' \in J_j$ is at a lower speed level in U or (b) any job $j' \in J_j$ is at a higher speed level outside of U. If that were not true, we get two (inactive) ϵ-transfers leaving U, one that can be lowered at its source (but is inactive because its destination is at an upper speed level) and one that can be increased at its destination (but is inactive because its source is at a lower speed level). Since both of these use j_a, we can concatenate them to get an active ϵ-transfer leaving U, contradicting our assumption. We now can simultaneously fix the speed levels of alls jobs $j' \in J_j$ by either (a) decreasing their speed levels $\mathcal{L}(j', \ell)$ for all $\ell \in \{ \ell_{\min}, \ell_{\min}+1, \dots, \ell_{\max} \}$ or (b) increasing their speed levels $\mathcal{L}(j', \ell)$ for all $\ell \notin \{ \ell_{\min}, \ell_{\min}+1, \dots, \ell_{\max} \}$. We iterate this until there are no more inactive ϵ-transfers leaving U.

It is easy to check that this procedure maintains Properties (a), (b), and (d) of the SLR. For Property (c), note that the described procedure (a) does not change the speed level difference between any two depletion intervals that are both inside or both outside of U and (b) any job that is active in- and outside of U will be part of some ϵ-transfer considered by the procedure, and thus its differences considered in Property (c) of the SLR decrease by exactly one at the left border of U and increase by exactly one at the right border of U. After fixing the speed levels, the set of nodes reachable from ℓ_0 has increased beyond U. This proves the lemma's statement. □

Lemma 100 shows how to continue to make progress in some situations. However there are still cases that don't allow progress with the current distribution graph but are not covered by that lemma. Also, there is a subtlety in the lemma's proof when we decrease speed levels. Consider a job j whose speed level in ℓ gets decreased. By Property (c), we have to make sure that, for any $\ell' < \ell$, the difference $\delta_{\ell,\ell'} := \mathcal{L}(j, \ell) - \mathcal{L}(j, \ell')$ remains non-negative (note that, by the same property, this value does not depend on j). To guarantee this, we merge any two depletion intervals as soon as the value $\delta_{\ell,\ell'}$ becomes zero.

It remains to show that if (after repairing speed levels and merging depletion intervals) we still cannot make progress, we actually found an optimal solution. We prove this in the following lemma.

Lemma 101. *Assume there is a depletion interval ℓ_0 for which there is no path of (active or inactive) ϵ-transfers to $L+1$. Then the current solution is optimal.*

Proof. We show the optimality of the current solution by showing that Property (d) of Theorem 68 holds. Together with the fact that our algorithm maintains Properties (a) to (c) of this theorem (see Theorem 99), this implies that the current solution is optimal.

To see that Property (d) holds, first notice that any job processed at a time $t \in I_\ell$ with deadline $d_j \in I_{\ell'}$ ($\ell' > \ell$) implies a (possibly inactive) ϵ-transfer of length one from ℓ to ℓ' (by just moving work of j). Now let ℓ_0 denote the rightmost depletion interval without a path of ϵ-transfers to $L+1$. Assume Property (d) is not true, so there is a job j with deadline $d_j > \tau_{\ell_0}$ and j is processed before τ_{ℓ_0}. Let $\ell_1 \leq \ell_0$ denote the depletion interval in which j is processed the first time and $\ell_2 > \ell_0$ the depletion interval that contains its deadline d_j. If $\ell_1 = \ell_0$, we're done: As noticed above, any such job would imply a (possibly inactive) ϵ-transfer of length one to ℓ_2. But then, as j was chosen maximal, there is a path from ℓ_2 to $L+1$. Together, these ϵ transfers form a path from ℓ_0 to $L+1$, a contradiction. So consider the case $\ell_1 < \ell_0$. Note that, without loss of generality, we can assume there is a job j' that is processed in ℓ_0 and has release time $r_{j'} < \tau_{\ell_0 - 1}$. If there is no such job, we could simply merge the depletion intervals ℓ_0 and $\ell_0 - 1$. Similar to the argument above, this job j' gives us a (possibly inactive) ϵ-transfer of length one from ℓ_0 to the left. By iterating this argument, we get a path of ϵ-transfers from ℓ_0 to ℓ_1 and, thus, to ℓ_2. As before, this yields a contradiction. $\square$

5.0 CONCLUSION

In this thesis, we provide a rigorous examination on the complexity of computing optimal schedules across a wide array of settings for speed scalable processors. There were four primary objectives of this thesis:

1: Determine the complexity across all thirty-two settings.

2: Extract broader complexity trends.

3: Develop our understanding of the combinatorial structure of optimal schedules.

4: Develop algorithmic tools to help us better reason about energy as a computational resource.

In Section 5.1 we discuss our progress vis--vis these objectives, and in Section 5.2 we discuss the remaining open problems as well as any insight this thesis may provide on them.

5.1 RESEARCH OBJECTIVES

5.1.1 Objective 1

The results of our first and second objective are captured in Table 2. As Table 2 shows, we resolved the individual complexity of all but four of the possible thirty-two settings. We discuss future approaches to resolving these remaining four settings below.

5.1.2 Objective 2

Beyond resolving individual complexities, our second objective aimed to utilize this taxonomy to capture complexity trends on a broader scale, thereby enabling practitioners to reason about the implications of their modeling and design decisions. One broader observation is that certain modeling decisions are sufficient for determining complexity. For example, any problem involving fractional flow, or any setting with identical jobs can be solved in polynomial time. In the former setting this generally follows from the ability to formulate the problem as a linear program whereas in the latter, the identical jobs give rise to simplified structures that one can leverage to yield polynomial time algorithms. On the contrary, problems involving integral flow in general prove to be more difficult. Aside from the simplified setting of identical jobs there are only hardness results.

Turning to the relationship between the energy budget and the flow plus energy problems, we see for every setting for which the complexity of each is known, the complexities (in terms of membership in P or NP-hardness) match, and furthermore we give explicit reductions between the two settings. So the takeaway here is that while modeling the performance objective as well as the workload can drive complexity, the decision of how we deal with the dual objectives is less relevant from a complexity standpoint. As we discuss below, it remains a possibility that one of the open problem will violate this general trend.

5.1.3 Objective 3

Despite improving our ability to reason about the structure of optimal schedules across several settings, many of these insights are difficult to crystalize in a compact statement. We provide here the insights that best capture the high level structure and that acted as guiding principals in our algorithm design. Beginning in Chapter 3, the primary structural takeaway is understanding how the optimal schedule for k jobs changes as a job $k+1$ of lower density is added. More precisely, the insight was understanding which of these first k jobs we need to increase in speed as we slowly increase the processing time given to job $k+1$. This structural insight is what allows us to build the schedule in an inductive manner.

The single most important structural insight from Chapter 4 is that when lowering the recharge rate, work tends to flow later in time in order for the schedule to remain feasible. Unfortunately this monotonic flow of work from left to right is not strict, which indeed is where much of the difficulty stems from. As a guiding principal however, determining paths on which work can flow towards the right is what allows us to use the homotopic approach of slowly lowering the recharge rate until reaching the optimal schedule.

5.1.4 Objective 4

Turning to our algorithmic contributions, we see that the general technique of formulating the given optimization problem as a mathematical program, subsequently using duality conditions to structurally characterize an optimal schedule, and finally using a homotopic approach to search for a schedule that satisfies this characterization, is a common theme in the polynomial time algorithms we develop. Both in Chapters 3 and 4, this general framework led us to polynomial time algorithms, and many of the polynomial time algorithms from Chapter 2 follow as a result of these. It is our hope that this approach will prove applicable in other energy aware optimization problems.

The second algorithmic tool, again seen both in Chapters 3 and 4, is the use of monotonicity in bounding the runtime of our algorithms. Not surprisingly, in order to get a grasp on bounding the number of events that force our homotopic search to pause and recompute, we must hone in on some property that is monotonic. In Chapter 3 this is relatively straightforward, a result of us considering jobs in a highest density first ordering. However as we see in Chapter 4, it was necessary to consider a nested structure of events in order to extract monotonicity properties. While this may lead to weaker bounds, again it is our hope that this nesting technique proves useful in more general settings.

5.2 OPEN QUESTIONS

Looking forward, there are two natural open questions from both the flow and deadline settings.

5.2.1 Open Problems for Flow Objective

In the flow setting, examining Table 2, we see we have resolved the complexity (in terms of membership in P or NP-hardness) of all settings except for `FE-IDUA`, `FE-ICUA`, `FE-IDWU`, and `FE-ICWU`. It is worth noting that given our reduction from the discrete case to the continuous case it is likely there are only really two open problems to resolve. That is, given our reduction, a polynomial time algorithm for the continuous case yields a polynomial time algorithm for the discrete case, and a hardness proof for the discrete case implies a hardness result for the continuous case. While it is possible that `FE-IDUA` is in P while `FE-ICUA` is NP-hard, this seems unlikely given that for every other setting in Table 2, the complexity of the discrete case and the continuous case match. We therefore examine only the discrete setting in our discussions below.

Question 1: Is FE-IDUA NP-hard or in P?

In this setting we are looking to minimize flow plus energy with integral flow, discrete speeds, unweighted jobs, and arbitrary sized work. The only evidence suggesting membership in P is that the problem of minimizing integral flow with unweighted jobs and arbitrary sized work on a single machine is in P. It is well known that the scheduling policy shortest remaining processing time first achieves optimality. However beyond this somewhat weak evidence, the existence of a polynomial time algorithm does not seem promising. Initial attempts proved difficult mainly a result of the fact that the structure of the schedule is highly sensitive to small perturbations of the instance. For example, slightly altering the amount of work for a single job can cause the completion ordering to change drastically. This seems to suggest that it is unlikely there are nice structural properties to guide the design and analysis of our algorithm.

Beyond our current inability to circumvent the structural issues, stronger evidence suggests that this problem is likely NP-hard. Looking at 2, we see that across all formulations where the complexity has been resolved, the complexity of the flow plus energy and budget problems are identical. This combined with the NP-hardness proof of B-IDUA suggest that if this trend holds true for all settings then FE-IDUA is NP-hard. Unfortunately, it is not clear how to establish a reduction from FE-IDUA to B-IDUA as we did in other settings, since here we will not have a FIFO completion ordering. So one insight here is that the same issue seems to be preventing both a positive and hardness analysis. Overcoming these obstacles will require new ideas in dealing with the fact that "similar" inputs can have vastly different completion orderings.

Question 2: Is FE-IDWU NP-hard or in P?

In this setting we are looking to minimize flow plus energy with integral flow, discrete speeds, weighted jobs, and unit sized work. While this setting is similar to `FE-IDWU` in the NP-hardness sense, there is one major difference which strongly suggests this problem is not in P. Whereas `FE-IDUA` is in P on a single machine without energy, minimizing integral flow on a single maching for weighted jobs and unit sized work has remained an open question for over 20 years. Since an algorithm for `FE-IDWU` would also yield an algorithm for the single speed case, resolving the complexity of this problem with a polynomial time algorithm will require new insights that have alluded researchers for the past 20 years. In terms of NP-hardness, we are in a similar situation as the previous setting where the reduction from the budget case does not seem to extend due to chaotic completion orderings.

5.2.2 Open Problems for Deadline Objective

Question 3: Can we find a polynomial time algorithm without the well separated assumption?

We present two major open problems related to deadline scheduling with a solar cell stemming from Chapter 4. One could argue the main caveat with this algorithm is the polynomial time runtime is predicated on the speeds being "well separated". While we provide argumentation why this is a realistic assumption, removing this assumption would strengthen the result and perhaps offer insight into improving the runtime.

Question 4: Can we find a polynomial time algorithm for a non-fixed recharge rate?

A second open problem would be to consider a slight variation on the optimization problem with a generalized recharge rate assumption. The variation is instead of minimizing the recharge rate, the recharge rate is fixed and we are tasked with finding the minimum energy feasible schedule for this fixed recharge rate. It is not terribly difficult to see that our algorithm actually solves this variation. The open problem comes by removing the assumption that the recharge rate is constant and instead considering piece-wise linear recharge rates. So while the former setting would be appropriate for say finding a minimum energy schedule for a single day (where the amount of sunlight is constant), the latter generalizes to finding optimal schedules for weather patterns emerging over a longer duration. It is not clear a priori how to generalize the homotopic approach presented in Chapter 4 when the recharge rate is no longer constant.

BIBLIOGRAPHY

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

[2] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms*, 3(4), 2007.

[3] Lachlan L. H. Andrew, Adam Wierman, and Ao Tang. Optimal speed scaling under arbitrary power functions. *SIGMETRICS Performance Evaluation Review*, 37(2):39–41, 2009.

[4] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Real-Time Systems, 13th Euromicro Conference*, pages 225–232, 2001.

[5] Hakan Aydin, Rami Melhem, Daniel Moss, and Pedro Meja-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. pages 95–105, 2001.

[6] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1):3:1–3:39, March 2007.

[7] Nikhil Bansal, Ho-Leung Chan, Tak Wah Lam, and Lap-Kei Lee. Scheduling for speed bounded processors. In *international conference on Automata, Languages, and Programming (ICALP)*, pages 409–420, 2008.

[8] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with a solar cell. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, AAIM '08, pages 15–26, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Nikhil Bansal, Kirk Pruhs, and Clifford Stein. Speed scaling for weighted flow time. *SIAM Journal on Computing*, 39(4):1294–1308, 2009.

[10] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. *ACM Transactions on Algorithms*, 9(2):18:1–18:14, 2013.

[11] Philippe Baptiste. Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. *Journal of Scheduling*, 2(6): 245–252, 1999.

[12] Philippe Baptiste. Scheduling equal-length jobs on identical parallel machines. *Discrete Appl. Math.*, 103(1-3):21–32, July 2000.

[13] Neal Barcelo, Daniel Cole, Dimitrios Letsios, Michael Nugent, and Kirk Pruhs. Optimal energy trade-off schedules. *Sustainable Computing: Informatics and Systems*, 3:207–217, 2013.

[14] Shiva Chaitanya, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Qdsl: A queuing model for systems with differential service levels. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 289–300, 2008.

[15] Sze-Hang Chan, Tak Wah Lam, and Lap-Kei Lee. Non-clairvoyant speed scaling for weighted flow time. In *European Symposium on Algorithms (ESA), Part I*, pages 23–35, 2010.

[16] Jian-Jia Chen and Chin-Fu Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 28–38, 2007.

[17] Jian-Jia Chen, Tei-Wei Kuo, and Chi-Sheng Shih. 1 + &#949; approximation clock rate assignment for periodic real-time tasks on a voltage-scaling processor. In *Proceedings of the 5th ACM International Conference on Embedded Software*, pages 247–250, 2005.

[18] Nikhil R. Devanur and Zhiyi Huang. Primal dual gives almost optimal energy efficient online algorithms. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1123–1140, 2014.

[19] A. Gandhi and M. Harchol-Balter. How data center size impacts the effectiveness of dynamic power management. In *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*, pages 1164–1169, 2011.

[20] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. Optimal power allocation in server farms. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, pages 157–168, 2009.

[21] Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. The case for sleep states in servers. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, pages 2:1–2:5, 2011.

[22] Jean-Baptiste Hiriart-Urruty and Claude Lemaréchal. *Fundamentals of Convex Analysis.* Springer, 2001.

[23] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 197–202, 1998.

[24] Woo-Cheol Kwon and Taewhan Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM Transactions on Embedded Computing Systems*, 4(1):211–230, 2005.

[25] Jacques Labetoulle, Eugene L. Lawler, Jan Karel Lenstra, and A. H. G. Rinnooy Kan. Preemptive scheduling of uniform machines subject to release dates. In Pulleyblank H. R., editor, *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, 1984.

[26] Tak Wah Lam, Lap-Kei Lee, Isaac Kar-Keung To, and Prudence W. H. Wong. Speed scaling functions for flow time scheduling based on active job count. In *European Symposium on Algorithms (ESA)*, pages 647–659, 2008.

[27] Minming Li and Frances F. Yao. An efficient algorithm for computing optimal discrete voltage schedules. *SIAM Journal of Computing*, 35(3):658–671, 2005.

[28] Nicole Megow and José Verschae. Dual techniques for scheduling on a machine with varying speed. In *Proceedings of the 40th International Conference on Automata, Languages, and Programming (ICALP)*, pages 745–756, 2013.

[29] Pedro Mejia-Alvarez, Eugene Levner, and Daniel Mossé. Adaptive scheduling server for power-aware real-time tasks. *ACM Transactions on Embedded Computer Systems*, 3(2): 284–306, 2004.

[30] Isi Mitrani. Managing performance and power consumption in a server farm. *Annals of Operations Research*, 202(1):121–134, 2013.

[31] Kirk Pruhs, Patchrawat Uthaisombut, and Gerhard J. Woeginger. Getting the best response for your erg. *ACM Transactions on Algorithms*, 4(3), 2008.

[32] Gang Quan and Xiaobo Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Design Automation Conference, 2001. Proceedings*, pages 828–833, 2001.

[33] Gang Quan and Xiaobo Sharon Hu. Minimum energy fixed-priority scheduling for variable voltage processors, 2002.

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