View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by D-Scholarship@Pitt

SPEED SCALING FOR ENERGY AWARE PROCESSOR SCHEDULING: ALGORITHMS AND ANALYSIS

by

Daniel Cole

B.S. in Computer Science and Engineering, The Ohio State University, 2003

Submitted to the Graduate Faculty of
The Dietrich School of Arts and Sciences in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Daniel Cole

It was defended on

April 2nd 2013

and approved by

Kirk Pruhs, Professor, Department of Computer Science

Rami Melhem, Professor, Department of Computer Science

Bruce Childers, Associate Professor, Department of Computer Science

Anupam Gupta, Associate Professor, Department of Computer Science, Carnegie Mellon University

Dissertation Director: Kirk Pruhs, Professor, Department of Computer Science

SPEED SCALING FOR ENERGY AWARE PROCESSOR SCHEDULING: ALGORITHMS AND ANALYSIS

Daniel Cole, PhD

University of Pittsburgh, 2013

We present theoretical algorithmic research of processor scheduling in an energy aware environment using the mechanism of speed scaling. We have two main goals in mind. The first is the development of algorithms that allow more energy efficient utilization of resources. The second goal is to further our ability to reason abstractly about energy in computing devices by developing and understanding algorithmic models of energy management. In order to achieve these goals, we investigate three classic process scheduling problems in the setting of a speed scalable processor.

Integer stretch is one of the most obvious classical scheduling objectives that has yet to be considered in the speed scaling setting. For the objective of integer stretch plus energy, we give an online scheduling algorithm that, for any input, produces a schedule with integer stretch plus energy that is competitive with the integer stretch plus energy of any schedule that finishes all jobs.

Second, we consider the problem of finding the schedule, $\mathcal{S}$, that minimizes some quality of service objective $\mathcal{Q}$ plus β times the energy used by the processor. This schedule, $\mathcal{S}$, is the optimal energy trade-off schedule in the sense that: no schedule can have better quality of service given the current investment of energy used by $\mathcal{S}$, and, an additional investment of one unit of energy is insufficient to improve the quality of service by more than β. When $\mathcal{Q}$ is fractional weighted flow, we show that the optimal energy trade-off schedule is unique and has a simple structure, thus making it easy to check the optimality of a schedule. We further show that the optimal energy trade-off schedule can be computed with a natural homotopic

optimization algorithm.

Lastly, we consider the speed scaling problem where the quality of service objective is deadline feasibility and the power objective is temperature. In the case of batched jobs, we give a simple algorithm to compute the optimal schedule. For general instances, we give a new online algorithm and show that it has a competitive ratio that is an order of magnitude better than the best previously known for this problem.

TABLE OF CONTENTS

LIST OF FIGURES

PREFACE

I would like to, first of all, thank my adviser, Kirk Pruhs, for being willing to be my adviser, for the various semesters of financial support, and for the actual time spent advising me. I would also like to thank my committee members, Bruce Childers, Rami Melhem, and Anupam Gupta, for taking time out of their busy schedules to serve on my committee.

1.0 INTRODUCTION

The development of new power aware technologies and the increased concern about the energy usage of computing devices presents new algorithmic challenges. New hardware mechanisms have been developed to allow more energy efficient hardware control, and whether the motivation is to prevent damage to hardware or the desire to limit the absolute energy usage, these new mechanisms require new algorithms in order to be as effective as possible. One of these new mechanisms is speed scaling technology. Speed scaling technology, which is now present in most desktop and laptop computers, allows a processor to run at different speeds, where each speed corresponds to a different power level. With the introduction of speed scaling, when an operating system schedules a process to run, in addition to having to choose which process to run, the operating system must, now, also decide at what speed to run the process. Should these two decisions (which process to run, and at what speed to run the process) be completely coupled, completely decoupled, or some balance of the two? Research into algorithmic power management seeks to answer these questions within a theoretical framework.

1.1 THE ALGORITHMIC PROBLEM CREATED BY SPEED SCALING

Speed scaling technology allows a processor to run at different speeds, where each speed corresponds to a different operating power. Although such a mechanism sounds simple enough, it is important to understand that speed scaling really only affects one component of the power of a processor, specifically, what is called the dynamic power of the processor.

The power of a processor is, generally, divided up into two components, static power

and dynamic power. The static power represents the rate at which the processor consumes energy because of the fact that the processor is in the "on" state, i.e. the rate of energy consumption when the processor is idle (on but not executing instructions). When the processor is executing instructions, the processor uses energy at a rate over and above the energy consumption rate of the processor when idle. This extra rate of energy usage is called the dynamic power. Because the mechanism of speed scaling deals with the speed of the processor, and thus the processor's power that is over and above the idle power, speed scaling affects only the dynamic power of the processor. Thus, in this work, when we refer to the power of a processor, unless explicitly stated, we mean the processor's dynamic power. As an example, if we say that a processor has 0 power, what we mean, specifically, is that the processor has 0 dynamic power. This is not to say that static power is not an important consideration with respect to the total energy usage of a processor; static power is an important consideration. However, because we are concerned with the mechanism of speed scaling, which deals with dynamic power, considering static power is outside of our scope.

Speed scaling technology presents so many new challenges in scheduling, not because the processor can now run at different speeds per se, but rather because, in general, running at a *higher* speed is *less* energy efficient. Thus, although running the processor at a higher speed will generally get jobs finished faster and thus improve quality of service, running the processor at a higher speed will also use more total energy.

A traditional estimate of the speed to power relationship of a processor is that the speed of a processor is proportional to the cube root of the dynamic power [13]. In general, the speed to power relation of a processor is increasing and strictly convex. It is this increasing and strictly convex property that gives rise to situations such as the one show in figure 1.1.

One motivation for speed scaling is to have a mechanism to help conserve energy, thus one of our goals is to use speed scaling to conserve or minimize energy. On the other hand, the point in considering scheduling algorithms at all is to enhance quality of service, thus our second goal is to maximize quality of service. But, as we have seen, in the speed scaling environment, we are faced with a situation where minimizing energy usage requires the slowest possible speed (and thus worst quality of service) and maximizing our quality of

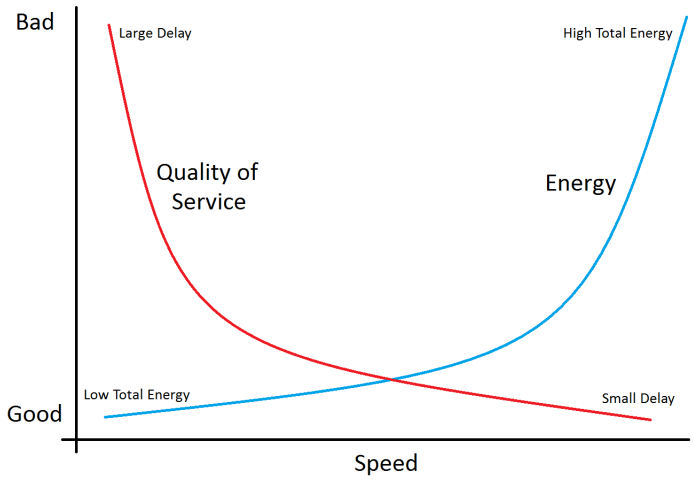

Figure 1: The x-axis is the constant speed used to run a fixed sized job. The red curve shows the delay of the job (time from release until completion) as we increase the speed and the blue curve shows the corresponding energy used to complete the job.

service requires the fastest possible speed (and thus largest amount of energy). Thus, our objectives are pushing against each other, they're in opposition to one another, and this is why speed scaling has introduced so many new algorithmic problems. We can't ignore either objective otherwise, as in figure 1.1, the objective we ignore can potentially reach unacceptable levels (i.e. very large energy usage or very poor quality of service). Instead, our goal is to determine how to schedule jobs such that the quality of service is good (fast speed) but the energy is also small (slow speed). However, what constitutes good quality of service or low energy usage is dependent on the situation and thus there is not a single right balance of the two objectives. Because of this, we need scheduling algorithms that, conceptually, have a dial adjustment. Turning the dial one way, causes the scheduler to favor quality of service more, and energy less, and turning the dial in the other direction causes the scheduler to favor energy more, and quality of service less. Thus the right balance, for any situation, can be achieved.

1.2 THE GOALS OF ALGORITHMIC ENERGY MANAGEMENT RESEARCH

There are two main goals in algorithmic research into energy management. The first is the development of algorithms that allow more efficient utilization of resources. For example, by developing a scheduling algorithm that minimizes the maximum temperature reached by a processor, more work can be done without having to increase cooling costs. By taking a mathematical approach, we can develop algorithms that are not necessarily intuitive or may not be found via trial and error.

The second main goal is to build our understanding of energy usage in computer systems in order to help us reason abstractly about energy and temperature. Toward this end, algorithmic research develops models that separate a mechanism from a particular device. For example, consider wireless network nodes that use a type of speed scaling in transmissions [19]. By modeling the speed scaling mechanism as an abstraction that holds for both types of devices, we can develop algorithms that work on both processors and wireless network links, even though the implementation of the mechanism varies between the devices. In this way, algorithmic energy management research aims to develop models that allow us to reason about energy and temperature in abstract computing devices in the same way we currently reason about space and time in abstract computing devices. Much in the same way that students are taught sorting algorithms without needing to reference whether the algorithm will run on a PC or a cell phone, we would like to be able to understand how to manage power with as little necessary hardware context as possible. However, power and energy, unlike space and time, do not appear to be inherent characteristics of computation as computation is reversible [12], and thus there appears to be no physical lower bound on the energy required to take one computational step (e.g. flipping a logical bit). In spite of this, it is important to pursue algorithmic research into power management as even a moderate level of abstraction can allow us to develop algorithms that apply to a large number of computing devices.

It is with these goals in mind that we seek to extend the theoretical understanding of algorithmic problems arising from the desire to directly limit or conserve energy usage

in computer systems. Specifically, we seek to extend current algorithmic understanding of how to use the new speed scaling hardware mechanism in the setting of classical process scheduling.

1.3 ALGORITHMIC MODELS FOR SPEED SCALABLE PROCESSORS

As previously mentioned, speed scaling requires the CPU scheduler to not only decide which job to run, but also how fast to run the job (whether to shut the processor off, and thus reduce or eliminate static power, is outside of our scope). Further, the speed at which a job runs need not be static, that is, the scheduler may change the running job's speed over time. However before we can develop new algorithms to make these decisions, we must first decide how to mathematically model a speed scalable processor. Modeling the processor requires that we define the set of allowable speeds as well as the set of possible power functions. We start by outlining the three standard speed models used in algorithmic speed scaling.

The first way to model the processor's speeds is with a discrete speed model. In this type of model the processor has a set speed values, each value specifying a single speed at which the processor can run. The second way to model the processor's speeds is with a continuous speed model. In the continuous speed model, the processor can run at a speed equal to any non-negative real value in some range, typically with a lower end of 0, i.e. idle. The upper end of the range may either be ∞, in which case there is, essentially, no upper limit on the speed, or some finite number, which means that there is a fixed upper limit on the speed. The last way to model the processor's speeds is with an arbitrary speed model which is a generalization of both the discrete and continuous speed models. The arbitrary speed model combines the discrete and continuous speed models by allowing the set of allowable speeds to be a set such that each item in the set is either a single discrete speed, or a single continuous interval. Thus each item in the set specifies a single speed at which the processor can run or an interval such that the processor may run at any speed in the interval. For each interval in the set, we assume that both ends of the interval are closed, the one exception being that there can be a single interval that is open on the upper end of ∞.

Turning to processor's power function, we also have three types of models. The first way to model power is with a discrete power model where the processor's possible powers are some discrete set of values. Specifically, this power model corresponds to the discrete speed model. When the discrete speed and power models are combined, the processor is described by a set of speed power pairs, such that, for each pair, the processor set to the given speed runs at the corresponding power. When designing an algorithm in this discrete speed/discrete power model, all we can assume is that the allowable speeds are a discrete set and that the speed to power relation is strictly convex and increasing. The appeal of the discrete model is that a processor model consisting of a finite set of speed/power pairs closely matches the operation of speed scalable processors, which typically may only be run at a finite set of speeds.

The second way to model a processor's power is with a continuous power model. In the continuous power model, the set of powers at which a processor can run is described by some continuous function, or some family of continuous functions. The functions are usually continuous on the interval $[0, \infty)$, and thus can be combined with the continuous speed model regardless of whether or not there is an upper bound on the possible speeds. An example of a power function in the continuous power model would be $P(s) = s^3$, meaning the power of the processor at speed s is s cubed. Because there is often no advantage, mathematically, to using s^3 versus, for example, s^4, the continuous power model is almost always generalized to say that, for example, the power function is of the form s^α for some $\alpha > 1$. In such a case, the algorithm designer does not know the specific value of α for some instance of the problem, however, when an algorithm is running it may have access to the value of α. Thus, in designing an algorithm we may incorporate α into the algorithm under the assumption that the scheduling algorithm will know the specific value of α at run time. The appeal of using a continuous power model is that continuous functions, such as s^α or 2^s, are usually easier to work with mathematically than a set of discrete power values.

The last way to model the processor's power is with an arbitrary power model which is allows the power function to be, essentially, anything. Specifically, there is some well defined function P such that the processor will run at power $P(s)$ when running at speed s. Similar to the continuous power model, the arbitrary power model is used such that the algorithm

designer does not know the specific P for an instance of the problem, but the algorithm, when running on the instance, will know P. Thus, the function P can be incorporated into the scheduling algorithm. The attraction of using the arbitrary power model is that it is the most general, but this fact is also what makes the arbitrary power model the most difficult to deal with when designing algorithms. Although, in general, we will be able to assume, without loss of generality, that P is increasing and strictly convex, P is still difficult to work with because we don't have any other properties of P or even a generalized closed form for P (we have a generalized closed form description for the power function in the continuous power model, e.g. s^α, ϵ^s). And, in fact, for some particular instance of a problem, P may not even have a closed form description. Further, for some problems, the best results that can be obtained are dependent on P and thus the arbitrary power model may only give results that are not meaningful and/or difficult to obtain because so few of the properties of P are known by the algorithm designer.

We now turn to extending the processor model to account for temperature. Note that when we consider the speed or power of a processor, while these values are measurable, more importantly, they are characteristics of a processor that we control directly. In other words, speed and power are both determined directly by configuration of the processor. When considering scheduling problems involving temperature, the model becomes more complicated because, while we can measure the instantaneous temperature of the processor, the temperature of a processor is not characteristic that we can set directly. The temperature of a processor is, rather, an indirect result of the configuration of the processor up to the current time. Thus, what we really need to model is, how does a processor's temperature behave as a function of its configuration (i.e. its power). One standard method to model the processor's change in temperature is by applying Newton's Law of Cooling.

A processor's change in temperature is dependent on how fast the processor releases, as heat, the energy it uses, as well as the rate at which it uses energy (i.e. the power). The rate at which the processor releases heat, i.e. how fast the processor cools, is dependent, not only on static characteristics of the processor itself (shape, etc.), but also the processor's environment and state (e.g. the temperature of the environment, the temperature of the processor, etc.). Cooling is, thus, a complex phenomenon that is difficult to model accurately. [10] suggested

assuming that all heat is lost via conduction, and that the ambient temperature is constant. This is not a completely unrealistic assumption, as the purpose of fans within computers is to remove heat via conduction, and the purpose of air conditioning is to maintain a constant ambient temperature. Newton's law of cooling states that the rate of cooling is proportional to the difference in temperature between the device and the ambient environment. This gives rise to the following differential equation describing the temperature T of a device as a function of time t:

$$\frac{dT(t)}{dt} = P(t) - bT(t)$$

That is, the rate of increase in temperature is proportional to the power $P(t)$ used by the device at time t (i.e. the rate at which energy is added to the processor), and the rate of decrease in temperature due to cooling is proportional to the temperature, or, more exactly, the difference in temperature between the processor and the environment. Because we assume the environment is at a constant temperature, we can translate the temperature scale so that the ambient temperature is 0. b is a device specific constant, called the *cooling parameter*, which describes how easily the device loses heat through conduction. For example, all else being equal, the cooling parameter would be higher for devices with high surface area than for devices with low surface area. Also note that this temperature model works regardless of the speed to power model of the processor.

1.4 MEASURING THE QUALITY OF ALGORITHMS

A schedule is defined as specifying, at all times, the job to be run at that time (including not running a job) and the speed at which to run the processor. We call a schedule feasible if the schedule does not violate any of the problem constraints (so feasibility is problem specific). For example, a feasible schedule will never specify that the processor be run at a speed that is not in the set of allowable speeds. Unless we are emphasizing the feasibility (or infeasibility) property of a schedule, when we talk about a schedule we mean a feasible schedule.

Once we have chosen a model for our speed scalable processor we need to be able to

measure the quality of a schedule when the schedule is run on the processor model. Classically, schedules have been measured by metrics such as the sum of the waiting time of all jobs (flow) and the number of jobs that meet their deadlines (throughput), as well as many others. We call these quality of service objectives. With a speed scalable processor, we still care about the quality of service of a schedule, but, now, we also care about some energy related objective, such as the total energy or maximum temperature. Having two objectives creates a new difficulty because our two objectives, high quality of service and low energy usage, are in conflict with one another. We could, of course, ignore, either quality of service or energy, and while this may be appropriate in some settings, it is clearly not acceptable in the extreme because it leads to either not running any jobs (ignore quality of service and minimize energy used) or using infinite energy or releasing more heat than the CPU can handle (ignore energy and optimize quality of service). Instead we would like to consider both energy and quality of service. We use two main methods to handle the conflict between energy and quality of services objectives.

The first method used to reconcile the opposing objectives of energy and quality of service is to combine energy and quality of service into a single objective function. Specifically, minimizing a linear combination $\alpha\mathcal{S}+\beta\mathcal{E}$ of a scheduling objective $\mathcal{S}$ and energy consumption $\mathcal{E}$ (for example, $\mathcal{S}$ could be total flow (wait time) and $\mathcal{E}$ could be total energy). α and β, then, allow the user to specify the relative importance of the scheduling objective and energy because the optimal schedule for the objective $\alpha\mathcal{S} + \beta\mathcal{E}$ is one that invests energy in such a way as to give the best resulting decrease in the scheduling objective $\mathcal{S}$ until the investment of an additional unit of energy cannot result in a schedule that reduces the scheduling objective by more than $\frac{\beta}{\alpha}$. A good schedule for a combined objective function must balance the two objectives (according to α and β which, by rescaling the units, we can assume are both 1). If the two objectives aren't balanced, then either energy cost or quality of service dominate the combined objective and, thus, very likely the total cost will be very large. Figure 1.4 illustrates this point by showing the objective of flow (wait time) plus energy for a single job as a function of the (fixed) speed at which the job is run.

The second method used is to fix either of the objectives at some predetermined level, and then focus on optimizing the other, non-fixed, objective. A problem where each job has

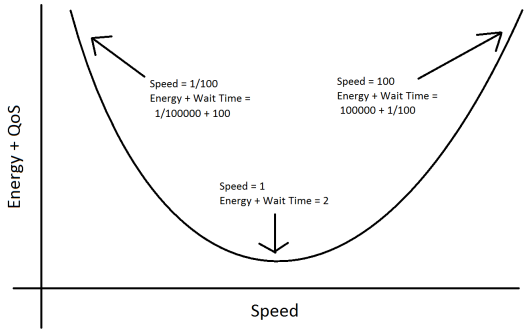

Figure 2: The x-axis is the fixed speed at which a job, of size 1, is run and the y-axis is the flow plus energy when power is speed cubed. At either extreme, only one of the objectives (either energy or flow) dominates the objective cost, but at the minimum, the costs are almost equal. Note that this curve is very similar to the sum of the two curves in figure 1.1.

a fixed deadline and we optimize the total energy used is an example of fixing one of the objectives and optimizing the other. This method is most often used for problems where there is some (known) fixed bound on one of the objectives that absolutely must not be violated.

If a schedule has the best objective value of any possible schedule, we say the schedule is optimal. Likewise, if an algorithm produces an optimal schedule for every possible input, we say the algorithm is optimal. However, very often, we will be considering online problems. In online problems, the scheduler does not become aware of a job until it is submitted to the system. In other words, the scheduler only knows the past and present, but not the future. Thus, in the online case, the scheduler must decide what job to run, and at what speed, at the current time without having complete information. This lack of complete information often means that no online scheduler can produce the optimal schedule for all possible inputs. In order to be able to measure the quality of an online scheduling algorithm, and the schedules the algorithm produces, we use worst case relative error. Worst case relative error is a guarantee of the quality of an algorithm for the case of the algorithm's worst input relative to any other algorithm. For online scheduling algorithms, a worst case

relative error guarantee states that, for any input, the schedule produced by the algorithm has an objective value that is no more than C times worse than the objective value of all possible schedules for that input (including the best possible schedule). When C is constant, i.e., not a function of the input size, we say that such an algorithm is "'competitive."' The appeal of a competitive online algorithm A, is that, for any input I, if A's schedule for I has high cost, then the optimal schedule for I also has high cost. In other words, if A's schedule has high cost, then the cost to schedule I is inherently high, that is, there is no schedule that makes the cost to schedule I low. On the other hand, if A's schedule for I has a low cost, then we're happy with the schedule because it has low cost.

1.5 INTEGRAL STRETCH PLUS ENERGY

Within the line of algorithmic speed scaling research that considers the objective of minimizing a linear combination $\alpha\mathcal{S} + \beta\mathcal{E}$ of a scheduling objective $\mathcal{S}$ and energy consumption, one natural question, for either clairvoyant or non-clairvoyant schedulers, is, "For what scheduling objectives $\mathcal{S}$ is there a speed scaling algorithm that is competitive for $\mathcal{S} + \mathcal{E}$ when the power function is arbitrary?". A competitive algorithm is achievable for some scheduling objectives. For example, for the scheduling objective of integer flow, there is a clairvoyant speed scaling algorithm that is constant competitive for an arbitrary power function [8]. This algorithm uses Shortest Remaining Processing Time (SRPT) for job selection and, at each time, chooses the processor speed that results in power consumption equal to the number of unfinished jobs. For the scheduling objective of fractional weighted flow, there is a clairvoyant speed scaling algorithm that has constant competitiveness for an arbitrary power function [8]. This algorithm uses Highest Density First (HDF) for job selection and, at each time, sets the processor speed so that the power consumption equals the total fractional weight of the unfinished jobs. Both of these speed scaling algorithms use the "natural" speed scaling algorithm that balances the rate of increase of the scheduling objective with the rate of increase of the energy objective by setting the instantaneous increase of the power consumption equal to the instantaneous increase of the scheduling objective $\mathcal{S}$.

For other scheduling objectives, constant competitiveness is only achievable when restrictions are placed on the growth rate of the power function P. For example, in [14], Chan *et al.* showed that non-clairvoyant speed scaling algorithms cannot achieve constant competitiveness for the objective of integer flow plus energy if the power function is too steep, although constant competitiveness is achievable if P is assumed to be bounded by a polynomial with a constant degree. They extended the previous work in [29] that showed there is no non-clairvoyant algorithm with constant competitiveness for the objective of integer flow time on a fixed speed processor. Given the evidence to date, in chapter 2, we make the following conjecture:

Conjecture. *There exists an online scheduling algorithm that is competitive on a fixed speed processor for the scheduling objective $\mathcal{S}$ if and only if there exists an online scheduling algorithm that is competitive for the scheduling objective $\mathcal{S}+\mathcal{E}$ when an arbitrary power function is considered.*

Unfortunately, it seems hard to imagine that the conjecture can be formally proven. A difficulty in showing the conjecture is that it does not seem possible to use the fact that competitiveness is achievable by an algorithm A for the scheduling objective $\mathcal{S}$ in the fixed speed setting as a black box to deduce anything about A in the speed scaling setting. Indeed, all previous analysis of algorithms in the speed scaling setting must know *why* A was competitive in the fixed speed setting. Therefore, to gain confidence of the truth of this conjecture perhaps one must settle for showing that the conjecture holds for particular natural scheduling objectives. One of the most obvious classical scheduling objectives that has yet to be considered in the speed scaling setting is integer stretch.

Integral flow time measures the time a job is delayed in the system, i.e. the time from the job's release until completion. The integral stretch of a job is the job's flow time divided by the size of the job. The integral stretch of the schedule is then the sum of all jobs' integral stretch. Another way to think of stretch is as a special case of the weighted flow metric (a job's weighted flow is the job's flow times the job's, input assigned, weight), specifically, where the weight of each job is the inverse of the job's size. The stretch metric measures how much the completion of a job was delayed relative to the completion time of the job

when running on a dedicated processor. Integer stretch may be a better quality of service metric when there are both small and large jobs as we expect the large jobs to take longer than the small jobs.

In the fixed speed setting, it is known that SRPT is competitive, but not optimal, for integer stretch [30], and in chapter 2, we extend this work by showing that there is a competitive algorithm for integer stretch in the speed scaling setting. In order to extend integer stretch to the speed scaling setting, we consider the classical scheduling input of a set of jobs with release times and sizes with the objective of minimizing the stretch plus energy of the schedule. We consider this problem when the processor can have an arbitrary speed to power function and give an algorithm, A, that is a competitive in the online model, that is, when the scheduler does not know either the existence or size of a job until the job's release time.

Our algorithm, A, uses a job selection algorithm that is a variation of Highest Density First, and a speed scaling algorithm that is a variation of the natural algorithm. Highest Density First schedules at time t the unfinished job with the largest weight divided by size (recall that the stretch of a job is equal to the weighted flow of a job when the job's weight is the inverse of its size). The "natural" speed scaling algorithm runs the processor at the speed that balances the current rate of increase of the scheduling objective with the current rate of increase of the energy objective by setting the instantaneous increase of the energy consumption equal to the instantaneous increase of the scheduling objective $\mathcal{S}$. One difficulty in analyzing the competitiveness of this algorithm is that there is no optimal online scheduling algorithm for the objective of integer stretch in the setting of a fixed speed processor [30]. The analysis in [8] of speed scaling algorithms for arbitrary power functions critically uses the fact that the scheduling algorithms considered (SRPT for the scheduling objective of integer flow and HDF for the scheduling objective of fractional weighted flow) are not only optimal, but have the additional property that they do not fall any further behind an arbitrary algorithm when starting from arbitrary configurations. Since we can not use the optimality of the scheduling algorithm in the fixed speed setting, our analysis is necessarily very different from the amortized local competitiveness analysis in [8]. By contrast, our analysis bounds the integer stretch of the algorithm A by its fractional stretch

and the aggregate integer stretch accumulated from jobs during times they are processed by A. We then bound these two quantities separately. For the scheduling objective of fractional weighted flow (a generalization of fractional stretch) plus energy, HDF is known to be competitive.The remaining cost of A, the integer stretch of the currently running job, we bound on a per job basis. Specifically, we bound a job's running integer stretch in A by the job's integer stretch in the optimal schedule and the energy the optimal schedule uses while running the job.

As an ancillary consequence of our analysis, we also show in chapter 2, that on a fixed speed processor, the integer stretch for SRPT is at most 3 times the fractional stretch of SRPT. This observation was the inspiration for our analysis of the speed scaling algorithm in chapter 2. It seems plausible that this, perhaps, mildly surprising relationship (this relationship doesn't hold for the flow objective, for example) between integer stretch and fractional stretch could prove useful in future research.

1.6 WEIGHTED FRACTIONAL FLOW PLUS ENERGY

Recall that the schedule, $\mathcal{S}$, that minimizes $\mathcal{Q}+\beta\mathcal{E}$, is the optimal energy trade-off schedule in the sense that, $\mathcal{S}$ minimizes $\mathcal{Q}$ subject to using total energy $\mathcal{E}$, and trading an additional unit of energy (increasing $\mathcal{E}$ by 1) cannot improve the quality of service by more than β (decrease $\mathcal{Q}$ by more than β). From this optimal trade-off point of view, some natural questions arise. Are there simple properties that characterize all optimal trade-off schedules? Are there multiple (fundamentally different) ways to optimally trade-off energy and quality of service? How does the optimal trade-off change as a function of β? In chapter 3, we seek to answer some of these types of questions.

Chapter 3 extends the work done in [31] where an efficient algorithm was given for the special case of unit jobs when the quality of service measure is total integer flow. The algorithm given in [31] can be thought of as a homotopic optimization algorithm. The setting for homotopic optimization is a collection of optimization problems with identical feasible regions, but with different objectives. If we have two objective functions $\mathcal{Q}$ and

$\mathcal{E}$, assume that when the objective is $\mathcal{Q}$, the optimization problem is easy to solve, but when the objective is $\mathcal{Q}+\mathcal{E}$, the optimization is not easy to solve. This is the case in the speed scaling setting because if energy is not part of the objective, the processor should always be run at maximum speed. Intuitively the homotopic optimization method solves the optimization problem, with objective $\mathcal{Q}+\mathcal{E}$, by maintaining the optimal solution with the objective $\mathcal{Q}+\beta\mathcal{E}$ as β continuously increases from 0 to 1. A highly desirable, if not strictly necessary condition, to develop a homotopic algorithm is that in some sense the optimal solution should change continuously as a function of β so that the knowledge of the optimal solution for a particular β is useful to obtain the optimal solution for $\beta+\epsilon$ for small ϵ. Thus a natural way to think about a homotopic optimization algorithm is to imagine the optimal solution being controlled by a slider that specifies β. As the slider is moved, β changes and thus the optimal solution evolves continuously with the slider.

[31] showed that if the quality of service measure is total flow, and jobs have arbitrary sizes, then the optimal energy trade-off schedule can change radically in response to a small change in β. Thus there is seemingly no hope of extending the work in [31] when the quality of service objective is total delay and jobs have arbitrary sizes. Because of this, we instead consider fractional flow, or more generally weighted fractional flow, and thus seek to characterize the schedule that minimize the total weighted fractional flow plus β times the energy used by the processor, i.e. the optimal trade-off schedule for fractional flow plus β times energy.

Recall that integral flow measures the time the job is delayed in the system, i.e. release to completion. Fractional flow weights this delay by the fraction of the job that is unfinished. For example, if a job is 60% done at a particular time, then the fractional flow of the job is increasing at a rate of 2/5 whereas the job's integral flow would still be increasing at a rate of 1. If each job is assigned a relative importance, i.e. a weight, the weighted fractional flow of a job is this weight times the job's fractional flow, thus the weighted fractional flow of a schedule is the sum of all jobs' weighted fractional flow. Fractional flow may be a preferred quality of service metric when the benefit of completing a fraction of a job is proportional to the fraction of the job that is completed. The input to the problem is a set of jobs where each job has a release time, a size, and a weight. We consider such inputs under the arbitrary

power function model.

Although we change the quality of service objective, we extend the general setting considered in [31] in three ways. First, we assume that jobs can have arbitrary sizes, as one would expect to be the case on a general purpose computational device. Second, we assume that each task has an associated importance, the previously mentioned weight of a job, derived from either some higher level application or from the user. Third, we assume that the allowable speeds, and the relationship between the speed and the power, of the processor are essentially arbitrary.

We first model the problem as a convex program and then apply the KKT conditions to derive necessary and sufficient conditions for the optimality of a schedule. We find that a feasible schedule is optimal if and only if three conditions hold:

- There is a linearly decreasing hypopower function associated with each job that maps a time to a hypopower, where the slope of the hypopower function is proportional to the density of the job, and inversely proportional to β. We coin the term hypopower to refer to the rate of change (derivative) of power with respect to speed as we are not aware of any standard name for this quantity.
- At all times, the job that is run is the job whose hypopower function is largest at that time.
- If a job is run, it is run at the hypopower specified by the hypopower function.

These conditions for optimal energy trade-off are very structurally different from the conditions for optimal energy trade-off when the quality of service objective is total flow. When the quality of service objective is total flow and all jobs have unit size, [31], showed that for any job i, if i doesn't delay other jobs then i is run as if alone, and if i does delay other jobs then i runs at a power equal to the power at which i would run if alone times the number of jobs i is delaying. While this is a slight simplification of the optimality conditions, we do see that the conditions of optimality are linear in power, while in our case, when the quality of service objective is fractional flow and jobs have arbitrary sizes, the conditions of optimality are linear in hypopower.

We then use these necessary and sufficient conditions to show additional facts about the

the optimal energy trade-off schedule. First, we show that any schedule may be checked for these conditions, and thus for optimality, in $O(n^2)$ time. Second, we show that as a consequence of these conditions, the optimal schedule is unique and that to specify an optimal schedule, it is sufficient to specify the value of each hypopower function at the release time of the corresponding job. Third, we show that a job's hypopower at release changes continuously as a function of β. Showing that a job's hypopower at release changes continuously is important because the time at which a particular bit of work is done does not change continuously as a function of β, as in [31]. Further, the fact that a job's hypopower at release changes continuously as a function of β allows us to develop an efficient homotopic algorithm for a modest number of jobs. The algorithm, which has a goal of finding the optimal energy trade-off schedule for a fixed $\beta > 0$, begins by creating the optimal energy trade-off schedule when $\beta = 0$ (which is trivial) and then maintains the optimal energy trade-off schedule as β increases to the desired value.

A trade-off objective that is commonly used outside of the theoretical speed scaling literature is to minimize the product of the quality of service and energy objectives. Thus, in chapter 3, we also consider the objective $\mathcal{Q}^\sigma \times \mathcal{E}$, where $\mathcal{Q}$ is fractional flow, $\mathcal{E}$ is energy, and σ is a parameter representing the relative importance of the scheduling objective and energy. We show the following:

- Perhaps counter-intuitively, we show that if the static power is zero, then the optimal solution is to either always go as fast as possible or to always go as slow as possible.
- We show that locally optimal product trade-off schedules have a nice structure similar to globally optimal sum trade-off schedules, but local optimality does not imply global optimality.
- We show that, for a fixed instance, the set of schedules that are optimal for the product objective (for any σ) are (a generally strict) subset of the schedules that are optimal for the sum objective (for any β).
- We show that the optimal product trade-off schedule may be a discontinuous function of σ. Thus there is unlikely to be a homotopic algorithm to compute optimal product trade-off schedules.

We therefore conclude that for the purposes of reasoning theoretically about optimal energy trade-off schedules, the sum trade-off objective is probably preferable to the product trade-off objective, as it has many more mathematically desirable properties.

1.7 MINIMIZING MAXIMUM TEMPERATURE UNDER DEADLINES

Recall that the other method to handle the conflicting objectives of energy and quality of service (as opposed to a linear combination of energy and quality of service) is to fix one of the objectives and optimize the other. We take this approach in chapter 4 by considering the classical deadline scheduling problem, that is the scheduling problem where each job has a deadline by which it must be completed. The problem has been extended to the speed scaling setting, by [33], by keeping the same input model (jobs have release times, sizes, and deadlines), and using the continuous speed model with power functions s^α for any speed $s \geq 0$. Note that in such a model, unlike the classical, non-speed scaling, setting, all inputs have feasible schedules because the processor can always run fast enough to get all jobs done by their deadlines. In chapter 4 we use this deadline model with an energy objective of minimizing the maximum temperature. With a power function of s^α, Newton's law of cooling gives the following differential equation to describe temperature:

$$\frac{dT(t)}{dt} = s(t)^\alpha - bT(t)$$

Where $T(t)$ is the temperature at time t, $s(t)$ is the speed at time t, and b is the processor specific cooling parameter.

A common online scheduling heuristic is to partition jobs into batches as they arrive. While a batch of jobs is being run, any jobs that arrive are collected in a new batch. When all jobs in the current batch are completed, a schedule for the new batched is computed and executed. We first consider the offline case of how to schedule jobs in this special batch case where all release times are zero.

For the offline case, we start with the feasibility problem for the batch case. The feasibility problem is to determine if the input has a feasible schedule (no deadlines violated) that

never exceeds a thermal threshold $T_{\max}$, which is also given as part of the input. We give a relatively simple $O(n^2)$ time algorithm. Our algorithm maintains the invariant that after the i^{th} iteration, it has computed a schedule S_i that completes the most work possible subject to the constraints that the first i deadlines are met and the temperature never exceeds $T_{\max}$. The main insight is that when extending S_i to S_{i+1}, one need only consider n possibilities, where each possibility corresponds to increasing the speed from immediately after one deadline before d_i until d_i in a particular way.

We then use the insights gained from solving the feasibility problem to solve the optimization problem, i.e., the problem of finding a deadline feasible schedule that minimizes the maximum temperature $T_{\max}$ attained. One obvious way to obtain an algorithm for this optimization problem would be to use the feasibility algorithm as a black box, and binary search over the possible maximum temperatures. This would result in an algorithm with running time $O(n^2 \log T_{\max})$. Instead we give an $O(n^2)$ time algorithm that in some sense mimics one run of the feasibility algorithm, raising $T_{\max}$ throughout so that it is always the minimum temperature necessary to maintain feasibility.

We then move on to dealing with the general online setting. Noting that it is perfectly reasonable that an operating system would have knowledge of the thermal threshold of the device on which it is scheduling tasks, we assume that the online algorithm knows the thermal threshold, T_{max}, of the device. We then give an online algorithm, A, that runs at a constant speed (a constant function of T_{max}) until an emergency arises, that is, it is determined that some job is in danger of missing its deadline. The speed in the non-emergency time is set so that in the limit the temperature of the device is at most a constant fraction of the thermal threshold. When an emergency is detected, the online algorithm A switches to using the OA speed scaling algorithm, which is guaranteed to finish all jobs by their deadline. When no unfinished jobs are in danger of missing a deadline, the speed scaling algorithm A switches from OA back to the non-emergency constant speed policy. We show that A is competitive for maximum temperature. The insight that allowed us to prove the competitiveness of A was that it is only necessary to run faster than A's constant speed for brief periods of time. In other words, any single emergency interval (when A is running OA) is short, specifically, the length of the emergency interval is bounded by a constant value that is proportional to

the inverse of the cooling parameter. We can then analyze the emergency and non-emergency periods separately.

2.0 SPEED SCALING FOR STRETCH PLUS ENERGY

In this chapter, we consider speed scaling problems where the objective is to minimize a linear combination of a scheduling objective $\mathcal{S}$ and the energy $\mathcal{E}$ used by the processor. These types of problems constitute a major line of algorithmic speed scaling research [3, 4, 7, 8, 11, 14, 15, 21, 22, 25, 26, 27]. A natural conjecture is that for any objective $\mathcal{S}$ there is an $O(1)$-competitive algorithm for $\mathcal{S}$ on a fixed speed processor if and only if there is an $O(1)$-competitive algorithm for $\mathcal{S} + \mathcal{E}$ on a processor with an arbitrary power function. We give evidence to support this conjecture by providing an $O(1)$-competitive algorithm for the objective of integer stretch plus energy. As an ancillary observation of this investigation, we show that on a fixed speed processor, the integer stretch for SRPT is at most 3 times the fractional stretch of SRPT.

2.1 RELATED WORK

[31] considered the problem of minimizing integer flow subject to an energy constraint, and gave an efficient offline algorithm for the case of unit size jobs. The problem of minimizing integer flow time plus energy was first proposed in [3]. They considered the unbounded speed scaling model with power function $P(s) = s^\alpha$, and gave an $O(1)$-competitive algorithm for the case of unit jobs. This was improved upon in [11] which showed that the natural speed scaling algorithm is 4-competitive for integer flow time plus energy for unit jobs. [11] further considered the objective of fractional weighted flow plus energy and gave an $O(\alpha/\log(\alpha))$-competitive algorithm which, using standard resource augmentation, implies an $O((\alpha/\log(\alpha))^2)$-competitive algorithm for integer weighted flow plus energy. [26] gave an

$O(\alpha/\log(\alpha))$-competitive algorithm for integer flow plus energy for arbitrary sized jobs. In the case that there is an upper bound on the speed of the processor, [7] gave an $O(\alpha/\log(\alpha))$-competitive algorithm for fractional weighted flow time plus energy for arbitrary sized jobs.

Speed scaling with arbitrary power functions was first considered in [8]. [8] shows that the speed scaling algorithm that runs at a power equal to 1 plus the number of active jobs, and that uses SRPT for job selection is a 3-competitive algorithm for the objective of integer flow plus energy. [8] also shows that the speed scaling algorithm that runs at power equal to the fractional weighted flow of all unfinished jobs, and that uses HDF for job selection is 2-competitive algorithm for the objective of fractional weighted flow plus energy. [4] shows that one could modify the analysis in [8] to show that the speed scaling algorithms that runs at a power equal to the number of unfinished jobs, and that uses SRPT for job selection is 2-competitive for the objective of integer flow plus energy. [4] further showed that no "natural" online speed scaling algorithm can be better than 2-competitive. [4] also showed that, for $P(s) = s^\alpha$ power functions, the natural speed scaling algorithm with processor sharing (PS) for job selection is $\max\{4\alpha - 2, 2(2 - 1/\alpha)^\alpha\}$-competitive. [21, 22] show how to extend the analysis in [8] to a setting of power heterogeneous multiprocessors.

2.2 PRELIMINARIES

An instance I consists of n jobs that arrive over time. Job i has release (arrival) time r_i, work/size p_i, and possibly a weight w_i. The density of a job is its weight divided by total size, that is w_i/p_i. A clairvoyant online scheduler is not aware of job i until time r_i, at which time it learns p_i and w_i. A non-clairvoyant scheduler does not learn p_i at time r_i. At each time the scheduler specifies a job to process and a speed for the processor. Preemption is allowed; that is, a job may be suspended and later restarted from the point of suspension. A job i is completed once p_i units of work have been processed on i. The speed is the rate at which work is processed. If job i of work p_i is run at a constant speed s until completion then job i will complete p_i/s time units after being started.

For a fixed schedule σ, we let C_i^σ denote job i's completion time. The flow (time)

$F_i^\sigma = C_i^\sigma - r_i$ of job i is the time that elapses after the job arrives until being completed. When the considered schedule σ is clear in the context, we may omit the superscript σ. Job i's integer stretch is F_i/p_i, and its integer weighted flow is $w_i F_i$. The integer stretch of a job can be viewed as a special case of integer weighted flow where the weight is the reciprocal of the job's size. The integer flow of a schedule is $\sum_{i=1}^n F_i$. The integer stretch of a schedule is $\sum_{i=1}^n F_i/p_i$, and the integer weighted flow is $\sum_{i=1}^n w_i F_i$. If the unfinished work of job i at time t is $p_i(t)$ then the fractional size of job i at time t is $p_i(t)/p_i$ and the fractional weight of job i at time t is $w_i(p_i(t)/p_i)$. The fractional flow of job i is defined to be $\int_{r_i}^\infty p_i(t)/p_i \, \mathtt{dt}$ and the fractional weighted flow of job i is $\int_{r_i}^\infty w_i(p_i(t)/p_i) \, \mathtt{dt}$. The fractional flow time and fractional weighted flow time of a schedule are the sum over all jobs of the fractional flow time or fractional weighted flow time respectively. The fractional stretch of a schedule is the same as the fractional weighted flow time where the weight of job i is set to $1/p_i$.

We adopt the speed scaling model originally proposed in [8] that essentially allows the power function to be arbitrary. In particular the power function may have a maximum power consumption rate, and hence maximum speed. The only real restriction on the power function is that it is piecewise continuous, differentiable, and integrable. It is shown in [8] that without loss of generality, we can assume that the power function $P : [0, s_{max}] \to [0, P_{max}]$ is continuous, differentiable, and convex, such that $P(0) = 0$ and $P(s_{max}) = P_{max}$. Here s_{max} (P_{max}) denote the maximum possible speed (power). By definition $P_{max} = P(s_{max})$. The energy (consumption) of a schedule is the power usage integrated over time.

We use $\widetilde{\text{WF}}$, S, $\widetilde{\text{S}}$, and E to denote the objectives of fractional weighted flow, integer stretch, fractional stretch, and energy, respectively. We use $\widetilde{\text{WF}}\oplus\text{E}$ and S$\oplus$E to denote the objective of fractional weighted flow plus energy and integer stretch plus energy, respectively. For an algorithm A we denote the schedule output by A on input I by $A(I)$. We use $OPT(I)$ to denote the optimal schedule for the objective under consideration. For example, $\widetilde{\text{WF}}\oplus\text{E}\langle A(I)\rangle$ denotes the fractional weighted flow plus energy for the schedule output by algorithm A on input I, and $S\oplus\text{E}\langle OPT(I)\rangle$ denotes the optimal integer stretch plus energy for input I.

2.3 AN ALGORITHM FOR STRETCH PLUS ENERGY WITH BOUNDED COMPETITIVENESS

In this section we give a clairvoyant speed scaling algorithm algorithm A, and show that A is $O(1)$ competitive for the objective of integer stretch plus energy on a processor with an arbitrary power function. We start by giving a definition of the online algorithm from [8], which we call BCP. BCP takes as input an online sequence of jobs with release times, sizes, and weights.

Definition of Online Algorithm BCP: At any time t, BCP always runs the unfinished job with the highest density at power

$$P_b(t) = min\{w_b(t), P_{max}\}$$

where P_{max} is the maximum power and $w_b(t)$ is the sum of the fractional weights of all unfinished jobs for BCP at time t.

We will assume that in case of equal density jobs, BCP breaks ties in favor of earlier released jobs. One of the two main theorems in [8] is Theorem 1, that BCP is 2-competitive for the objective of fractional weighted flow plus energy.

Theorem 1. *[8] For all inputs I, $\widetilde{WF}\oplus E\langle BCP(I)\rangle \leq 2\cdot \widetilde{WF}\oplus E\langle OPT(I)\rangle$.*

At a high level, A is the same as BCP with two important differences. The first difference is that A rounds the weight of each job up so that the density is an integer power of some constant $f > 1$, so that the following inequality holds

$$\frac{1}{p_j} \leq w_j < \frac{f}{p_j}$$

This is to prevent the algorithm A from preempting between jobs of similar densities. The second difference is that A never lets the power fall below one over the work of the active job. Let I be an arbitrary instance of jobs with release times and sizes. Let I' be the corresponding instance where additionally all jobs j have weights w_j, where w_j is the minimal real number, not less than $1/p_j$, such that w_j/p_j is an integer power of f. The algorithm A can either

be thought of as running on an unweighted instance, where the algorithm instantiates the weights, or on a weighted instance in which weights are provided as part of the input.

Definition of Online Algorithm A: When considering an unweighted instant I, A assigns jobs the weights that they would have in instance I'. At any time t, A runs the highest density job, breaking ties in favor of earlier released jobs, at power

$$P_a(t) = \min\left\{\max\left\{w_a(t), \frac{1}{p_{a(t)}}\right\}, P_{max}\right\}$$

where $a(t)$ is the job being processed at time t, $p_{a(t)}$ is the size of $a(t)$, and $w_a(t)$ is the sum of the fractional weights of all unfinished jobs at time t.

We show that, by picking $f \approx 2.015$, A is approximately 9.414-competitive for the objective of integer stretch plus energy. Our analysis can be summarized as the following sequence of bounding steps:

$$\begin{aligned}
\mathrm{S} \oplus \mathrm{E}\langle \mathrm{A}(I)\rangle &\leq \widetilde{\mathrm{WF}} \oplus \mathrm{E}\langle \mathrm{A}(I')\rangle + \left(1 + \frac{\sqrt{f}}{\sqrt{f}-1}\right)\int_t \frac{1}{p_{a(t)}} && \text{(Lemma 2)}\\
&\leq \widetilde{\mathrm{WF}} \oplus \mathrm{E}\langle \mathrm{A}(I')\rangle + \left(1 + \frac{\sqrt{f}}{\sqrt{f}-1}\right)\mathrm{S} \oplus \mathrm{E}\langle OPT(I)\rangle && \text{(Lemma 3)}\\
&\leq (2f+1)\cdot \mathrm{S} \oplus \mathrm{E}\langle OPT(I)\rangle + \left(1 + \frac{\sqrt{f}}{\sqrt{f}-1}\right)\mathrm{S} \oplus \mathrm{E}\langle OPT(I)\rangle && \text{(Lemma 4)}\\
&= \left(2f + 2 + \frac{\sqrt{f}}{\sqrt{f}-1}\right)\mathrm{S} \oplus \mathrm{E}\langle OPT(I)\rangle\\
&\approx 9.414 \cdot \mathrm{S} \oplus \mathrm{E}\langle OPT(I)\rangle
\end{aligned}$$

Before proving each of the lemmas used above, we give some intuitive explanation of the lemmas. Lemma 2 shows that the integer stretch of A is bounded by the fractional weighted flow of A plus the aggregate integer stretch accumulated from jobs while they are run by A. The proof relies on the fact that densities of the preempted jobs in A's schedule form a geometric sequence. Lemma 3 shows via a charging scheme that the aggregate integer stretch accumulated from jobs while they are run by A is at most the optimal integer stretch plus energy. Intuitively, this lemma can be explained as follows. For each job j, consider the times that j is processed by A and OPT. If OPT processes j faster than A by using more power, j's integer stretch can be charged to OPT's power usage. Otherwise, job j, as

an active job, contributes to A's integer stretch less than OPT's integer stretch. Lemma 4 relates the fractional weighted flow plus energy for A to the optimal integer stretch plus energy. Recall that A uses the same speed scheduling policy as BCP except that it runs at power at least $1/p_{a(t)}$ at any time t. Let us call the time periods that A uses power exactly $1/p_{a(t)}$ as minimum power periods and the other time periods as normal time periods. By mimicking the analysis of BCP, we can bound the fractional flow for A, plus the energy used by A during normal time periods. We separately bound the energy used by A during minimum power periods by the integer stretch of the active jobs. Before preceding to these lemmas, in Lemma 1 we make an intuitive observation that the optimal fractional weighted flow for instance I' is at most f times the optimal integer stretch plus energy for instance I.

Lemma 1. $\widetilde{WF} \oplus E\langle OPT(I')\rangle \leq f \cdot S \oplus E\langle OPT(I)\rangle$.

Proof. First note that

$$\widetilde{\mathrm{WF}} \oplus \mathrm{E}\,\langle OPT(I')\rangle \leq f \cdot \widetilde{\mathrm{S}} \oplus \mathrm{E}\,\langle OPT(I)\rangle$$

since each weight w_j in I' is at most f/p_j. Further it follows that

$$\widetilde{\mathrm{S}} \oplus \mathrm{E}\,\langle OPT(I)\rangle \leq \mathrm{S} \oplus \mathrm{E}\,\langle OPT(I)\rangle$$

since the fractional stretch of any schedule is no more than the integer stretch of that schedule. □

Lemma 2. $S \oplus E\langle A(I)\rangle \leq \widetilde{WF} \oplus E\langle A(I')\rangle + \left(1 + \frac{\sqrt{f}}{\sqrt{f}-1}\right) \int_t \frac{1}{p_{a(t)}}$.

Proof. Throughout this lemma we will implicitly use the fact that A creates the same schedule for the instance I and the instance I' by definition of A. To show the lemma, we focus on showing the stronger statement that

$$\mathrm{S}\,\langle A(I)\rangle \leq \widetilde{\mathrm{WF}}\,\langle A(I')\rangle + \left(1 + \frac{\sqrt{f}}{\sqrt{f}-1}\right) \int_t \frac{1}{p_{a(t)}}$$

Notice that this is strictly a stronger statement because the energy used by A is the same for I' and I. Fix a time t. We partition the unfinished jobs in A's schedule at time t into three sets: the running job, $a(t)$, the set $A_u(t)$ of jobs that have not been processed by A,

and the set $A_p(t)$ of jobs that have been partially processed by A, excluding the running job $a(t)$. It is sufficient to establish the following invariant:

$$\sum_{i \in A_u(t)} \frac{1}{p_i} + \sum_{i \in A_p(t)} \frac{1}{p_i} + \frac{1}{p_{a(t)}} \leq \sum_{i \in A_u(t)} w_i \frac{p_i(t)}{p_i} + \left(1 + \frac{\sqrt{f}}{\sqrt{f}-1}\right) \frac{1}{p_{a(t)}}.$$

Proving this invariant is sufficient to prove the lemma because the left hand side of the inequality is the instantaneous increase in the stretch objective for A's schedule at time t and

$$\sum_{i \in A_u(t)} w_i \frac{p_i(t)}{p_i}$$

is the instantaneous increase in the fractional weighted flow time of A's schedule at time t. Thus, if this inequality can be shown for all times t, then by integrating over time the lemma follows.

By definition $p_i(t)/p_i = 1$ for unprocessed jobs. Also, $w_i \geq 1/p_i$ by the definition of I'. Knowing this, we have

$$\sum_{i \in A_u(t)} \frac{1}{p_i} \leq \sum_{i \in A_u(t)} w_i \frac{p_i(t)}{p_i}.$$

Thus to establish our invariant, it is sufficient to show that

$$\sum_{i \in A_p(t)} \frac{1}{p_i} \leq \frac{\sqrt{f}}{\sqrt{f}-1} \frac{1}{p_{a(t)}}.$$

Consider the jobs $d(0), \ldots d(m)$ in $A_p(t)$ by increasing order of arrival time where $m = |A_p(t)|$. By the definition of I', any two jobs either have equal densities or their densities differ by at least a factor of f. Knowing that all jobs in $A_p(t)$ have been partially processed by A and that A breaks ties in favor of jobs with earlier release dates, it must be the case that

$$f \frac{w_{d(i)}}{p_{d(i)}} \leq \frac{w_{d(i+1)}}{p_{d(i+1)}}$$

for $0 \leq i \leq m-1$ and

$$f \frac{w_{d(m)}}{p_{d(m)}} \leq \frac{w_{a(t)}}{p_{a(t)}}$$

Hence we have

$$\frac{w_{d(i)}}{p_{d(i)}} \leq \frac{1}{f^{m-i+1}} \frac{w_{a(t)}}{p_{a(t)}}$$

For any job j, by definition of w_j, $\frac{1}{p_j} \leq w_j < \frac{f}{p_j}$. Knowing this, we have that

$$\begin{aligned}\frac{1}{(p_{d(i)})^2} &\leq \frac{w_{d(i)}}{p_{d(i)}} \\ &\leq \frac{1}{f^{m-i+1}} \frac{w_{a(t)}}{p_{a(t)}} \\ &\leq \frac{1}{(p_{a(t)})^2}\end{aligned}$$

Thus we have,

$$\begin{aligned}\sum_{i \in A_p(t)} \frac{1}{p_i} &= \sum_{i=0}^{m} \frac{1}{p_{d(i)}} \\ &\leq \sum_{i=0}^{m} \frac{1}{(\sqrt{f})^{m-i}} \frac{1}{p_{a(t)}} \\ &\leq \frac{1}{p_{a(t)}} \sum_{i=0}^{\infty} \frac{1}{(\sqrt{f})^{i}} \\ &= \frac{\sqrt{f}}{\sqrt{f}-1} \frac{1}{p_{a(t)}}\end{aligned}$$

□

Lemma 3. $\int_t \frac{1}{p_{a(t)}} \leq S \oplus E\langle OPT(I) \rangle$.

Proof. Consider any arbitrary job j with total work p_j, and an arbitrary infinitesimal portion of that work dp_j. Let dt_A be the amount of time that A spends actively working on p_j to complete the dp_j portion of p_j's work, and let dt_O be the amount of time that OPT spends actively working on p_j to complete the dp_j portion of p_j's work. We call dt_A/p_j the contribution of dp_j to $\int_t \frac{1}{p_{a(t)}}$. Our goal is to bound the contribution of dp_j by the optimal solution's cost when the optimal solution processes the dp_j portion of p_j's work. Once this is shown, by integrating over all portion's of p_j's work and summing over all jobs, the lemma follows. We consider two cases depending on the relationship of dt_A and dt_O.

First consider the case where $dt_A \leq dt_O$. In this case, we can charge the contribution of dp_j, to the the integer stretch penalty OPT incurs while processing the dp_j portion of p_j. Indeed, the integer stretch penalty OPT incurs for job j is dt_O/p_j during this time. Now consider the other case where $dt_A > dt_O$. In this case, the convexity of the power function

implies that A uses no more energy than OPT while processing the dp_j portion of p_j. In particular, A's energy usage is strictly smaller P_{max}. Thus, by definition of A, A runs the processor using power at least $1/p_j$ the entire time the dp_j portion of p_j is being processed. This implies that the energy used by A to complete the work dp_j is at least the integer stretch penalty dp_j/p_j incurred by A. Hence we can charge dp_j/p_j to the energy that OPT uses to process dp_j. □

Lemma 4. $\widetilde{WF} \oplus E\langle A(I')\rangle \leq (2f+1) \cdot S \oplus E\langle OPT(I)\rangle$.

Proof. For instances where densities are integer powers of f, the only difference between the algorithm A and the algorithm BCP is that on some configurations A would run faster than BCP. In particular, at times when $w_a(t) < 1/p_{a(t)} \leq P_{max}$, A will run at power $1/p_{a(t)}$ while BCP would only run at power $w_a(t)$. Recall that these times are called the minimum power periods for A.

The analysis of BCP in [8] uses an amortized local competitiveness argument. That is, it gives a potential function $\Phi(t)$ so that the following invariant holds at all times t:

$$w_a(t) + P_a(t) + \frac{d\Phi(t)}{dt} \leq 2(w_o(t) + P_o(t))$$

where $P(t)$ is the power used by A, $w_o(t)$ is the unfinished fractional weight for the optimal schedule and $P_o(t)$ is the power used in the optimal schedule. This invariant establishes that for BCP,

$$\widetilde{\mathrm{WF}} \oplus \mathrm{E}\langle \mathrm{BCP}(I')\rangle \leq 2 \cdot \mathrm{WF} \oplus \mathrm{E}\langle OPT(I')\rangle$$

Hence by Lemma 1,

$$\widetilde{\mathrm{WF}} \oplus \mathrm{E}\langle \mathrm{BCP}(I')\rangle \leq 2f \cdot \mathrm{S} \oplus \mathrm{E}\,\langle OPT(I)\rangle$$

If we attempted to repeat this analysis with the algorithm A, the only problem is that during the minimum power periods, A might run faster than BCP, meaning that this analysis would not account for all the energy used by A during the minimum power periods. Therefore, the only task left is to bound the total power used by A during the minimum power periods. Note that any time t that is a minimum power period, A uses power $1/p_{a(t)}$. Therefore, the total energy consumption by A during all the minimum power periods is bounded by

$\int_t 1/p_{a(t)}$. By Lemma 3, it is at most $\mathrm{S} \oplus \mathrm{E}\,\langle OPT(I)\rangle$. Combining this quantity with the upper bound obtained following the BCP analysis completes the proof. □

2.4 INTEGRAL STRETCH OF SRPT

In this section, we show that the integer stretch of SRPT is at most 3 times the fractional stretch of SRPT under constant speed curves.

Lemma 5. *$S\langle SRPT(I)\rangle \leq 3 \cdot \widetilde{S}\langle SRPT(I)\rangle$ for fixed speed processors.*

Proof. Without loss of generality we can assume the speed of the processor is 1 by rescaling the units of time. Let $c > 1$ be any constant, which will be fixed soon. Define $C^+(t)$ as the set of unfinished jobs at time t, excluding the running job, that have at least a $1/c$ fraction of their work processed by SRPT at time t. Let $C^-(t)$ be the set of unfinished jobs at time t, excluding the running job $a(t)$, have at most a $1/c$ fraction of their work processed by SRPT at time t. Formally,

$$C^+(t) := \{j \in A_u(t) \cup A_p(t) \mid \frac{p_j(t)}{p_j} \leq \frac{c-1}{c}\}$$

and

$$C^-(t) := A_u(t) \cup A_p(t) \setminus C^+(t)$$

We first show that the total accumulated stretch for the active jobs is at most twice their total accumulated fractional stretch:

$$\int_t \frac{1}{p_{a(t)}} \leq 2 \int_t \frac{p_{a(t)}(t)}{p_{a(t)}} \frac{1}{p_{a(t)}} \tag{2.1}$$

We will also show the following invariants hold at all times:

$$\sum_{i \in C^+(t)} \frac{1}{p_i} \leq (c-1)\frac{1}{p_{a(t)}} \tag{2.2}$$

and

$$\sum_{i \in C^-(t)} \frac{1}{p_i} \leq \frac{c}{c-1} \sum_{i \in C^-(t)} \frac{p_i(t)}{p_i} \frac{1}{p_i} \tag{2.3}$$

By definition of the fractional stretch objective, the stretch objective, equation (2.1) and equation (2.2), we have that

$$\begin{aligned}\int_t \sum_{i \in C^+(t) \cup \{a(t)\}} \frac{1}{p_i} &\leq \int_t (c-1)\frac{1}{p_{a(t)}} + 2\int_t \frac{p_{a(t)}(t)}{p_{a(t)}}\frac{1}{p_{a(t)}} \\ &\leq (2(c-1)+2)\int_t \frac{p_{a(t)}(t)}{p_{a(t)}}\frac{1}{p_{a(t)}} \\ &= 2c\int_t \frac{p_{a(t)}(t)}{p_{a(t)}}\frac{1}{p_{a(t)}}\end{aligned} \tag{2.4}$$

Combining inequality (2.3) and inequality (2.4), we can conclude that

$$\mathrm{S}\langle SRPT(I)\rangle \leq \max\left\{2c, \frac{c}{c-1}\right\} \cdot \widetilde{\mathrm{S}}\langle SRPT(I)\rangle$$

The tightest bound is achieved when $c = 3/2$, which gives the desired bound that

$$\mathrm{S}\langle SRPT(I)\rangle \leq 3 \cdot \widetilde{\mathrm{S}}\langle SRPT(I)\rangle$$

We now prove equation (2.1). It is sufficient to show that the integer stretch incurred for a job i, while i is running, is at most twice the fractional stretch incurred during the times that i is running. Since we only care about the times when i is running, without loss of generality we can simply consider these times as one contiguous period of time of length p_i. The integer stretch incurred is,

$$\int_0^{p_i} \frac{1}{p_i}\mathtt{dt} = 1$$

and the fractional stretch is exactly half of this, namely

$$\int_0^{p_i} \left(\frac{p_i - t}{p_i}\right)\frac{1}{p_i}\mathtt{dt} = \frac{1}{2}$$

We now prove equation (2.2). Consider the jobs in $C^+(t) \cup \{a(t)\}$, indexed from $d(0)$ to $d(m) = a(t)$ increasing order of release time. Recall that jobs in $C^+(t)$ have at least a $\frac{1}{c}$ fraction of the processing time completed. Since SRPT processed job $d(i)$ over $d(i-1)$ with remaining work at most

$$\left(1 - \frac{1}{c}\right) p_{d(i-1)}$$

it must be the case that

$$\left(1 - \frac{1}{c}\right) p_{d(i-1)} \geq p_{d(i)}$$

for $1 \leq i \leq m$. By expanding this recurrence we obtain that

$$\frac{1}{p_i} \leq \left(\frac{c-1}{c}\right)^{m-i} \frac{1}{p_{a(t)}}$$

, for $0 \leq i \leq m$. Thus we have

$$\begin{aligned}
\sum_{i \in C^+(t)} \frac{1}{p_i} &= \sum_{i=0}^{m-1} \frac{1}{p_{d(i)}} \\
&\leq \frac{1}{p_{a(t)}} \sum_{i=0}^{m-1} \left(\frac{c-1}{c}\right)^{m-i} \\
&= \frac{1}{p_{a(t)}} \sum_{i=1}^{m} \left(\frac{c-1}{c}\right)^{i} \\
&\leq \frac{1}{p_{a(t)}} \sum_{i=1}^{\infty} \left(\frac{c-1}{c}\right)^{i} \\
&= \frac{c-1}{p_{a(t)}}
\end{aligned}$$

We finish the proof by noting that equation (2.3) follows immediately from the fact that

$$\frac{p_i(t)}{p_i} \geq \frac{c-1}{c}$$

for all jobs in $C^-(t)$. □

It might be interesting to determine if one can obtain a tight bound on the worst case ratio between the integer stretch and the fractional stretch for SRPT. One can construct instances where the ratio is strictly greater than two.

3.0 OPTIMAL ENERGY TRADE-OFF SCHEDULES

In this section, we consider scheduling tasks that arrive over time on a speed scalable processor. At each time a schedule specifies a job to be run and the speed at which the processor is run. We seek to understand the structure of schedules that optimally trade-off the energy used by the processor with a common scheduling quality of service measure, fractional weighted delay. We assume that there is some user defined parameter β specifying the user's desired additive trade-off between energy efficiency and quality of service. We prove that the optimal energy trade-off schedule is essentially unique, and has a simple structure. Thus it is easy to check the optimality of a schedule. We further prove that the optimal energy trade-off schedule changes continuously as a function of the parameter β. Thus it is possible to compute the optimal energy trade-off schedule using a natural homotopic optimization algorithm. We further show that multiplicative trade-off schedules have fewer desirable properties.

3.1 RELATED WORK

The work in [31], as presented, assume an objective of total delay subject to a constraint on the total energy used, although it is straight-forward to see that the same approach works when the objective is a linear combination of total delay and energy. [3] introduced the idea of considering an objective that is a linear combination of energy and a scheduling quality of service objective into the literature. [3] gave a dynamic programming based algorithm to compute the optimal energy trade-off schedule for unit work jobs. To the best of our knowledge there is no other algorithmic work directly related to the work in this paper.

[8] showed that a natural online algorithm is 2-competitive for the objective of a linear combination of energy and weighted fractional delay. [8, 5] showed that a natural online algorithm is 2-competitive for the objective of a linear combination of energy and total (unweighted) (integer) delay. Previously, [3, 11, 26, 14, 17, 18, 16] gave online algorithms with competitive analysis in the case that the power function was of the form s^α.

[23] first introduced the energy-delay product as a metric, and its use has been prevalent since then.

3.2 PRELIMINARIES

The input consists of n jobs, where job i has release time r_i, work p_i, and a weight $w_i > 0$. The density of a job is its weight divided by work, that is w_i/p_i. A schedule is defined by specifying, for each time, a job to be run and a speed at which to run the processor. Preemption is allowed, that is, a job may be suspended and later restarted from the point of suspension. A schedule may only specify a job i to be run at time t if i has been released by t, i.e., $t \geq r_i$. Job i is completed once p_i units of work have been performed on i. Speed is the rate at which work is completed, thus if job i, of work p_i, is run at constant speed s until completion, job i will complete p_i/s time units after being started. Without loss of generality, we assume that no two jobs are released at the same time. If this is not the case, then we can create a new input that spaces out the releases of jobs released at the same time by $\epsilon > 0$, where jobs are then released in order of non-increasing w_i/p_i. An optimal schedule for this new input is optimal for the original input as the optimal schedule for the original input works on each job for at least time ϵ and prioritizes jobs by non-increasing w_i/p_i.

For job i, the delay (also called flow), F_i, is i's completion time minus its release time and the weighted delay is $w_i F_i$. The delay of a schedule is $\sum_{i=1}^n F_i$ and the weighted delay is $\sum_{i=1}^n w_i F_i$. If $p_i(t)$ work is performed on job i at time t then a $p_i(t)/p_i$ fraction of job i was delayed $t - r_i$ time units before being completed, thus the total fractional delay of job i is $\int_{r_i}^\infty p_i(t)(t - r_i)/p_i \, dt$ and the fractional weighted delay of job i is $\int_{r_i}^\infty w_i p_i(t)(t - r_i)/p_i \, dt$. The fractional delay and fractional weighted delay of a schedule are the sum, over all jobs,

of the fractional delay and fractional weighted delay respectively. The objective we consider is the energy used by the schedule plus the fractional weighted delay of the schedule.

We adopt the speed scaling model, originally proposed in [8], that essentially allows the speed to power function to be arbitrary. In particular the power function may have a maximum power, and hence maximum speed. For our setting, [8] showed that results that hold for power functions that are continuous, differentiable, and convex from speed/power 0 to the maximum speed/power, will hold for any power function meeting the less strict constraints of being piecewise continuous, differentiable, and integrable. Thus, without loss of generality, we may assume the power function is continuous, differentiable, and convex from speed/power 0 to the maximum speed/power. We define the power function as $P(S)$, where S is a speed. Energy is power integrated over time, $\int_t P(S)dt$. The static power is equal to $P(0)$, which is the power used by the processor even if it is idling.

We use the term hypopower to refer to the derivative of power with respect to speed. A particular hypopower function that will be important to our discussions is the derivative of the processor's power function, $P(S)$, with respect to speed, which will be denoted as $P'(S)$. The functional inverse of $P'(S)$ will be denoted as $P^*(x)$, where x is a hypopower and $P^*(x)$ is speed. It's important to keep in mind that the definition of hypopower is completely unrelated to the function $P'(S)$ in the same way that the definition of power is completely unrelated to the function $P(S)$.

In the context of schedules we will need to specify how speed, power, and hypopower change over time, thus we define $S(t, A)$, $P(t, A)$, and $P'(t, A)$, as the speed, power, and hypopower respectively, of schedule A at time t, such that $P(t, A) = P(S(t, A))$ and $P'(t, A) = P'(S(t, A))$. If A is understood, then we drop A. Further, we denote the speed, power, and hypopower experienced by a job i run at a time t in a schedule A as $S_i(t, A)$, $P_i(t, A)$, and $P'_i(t, A)$ respectively, such that $P_i(t, A) = P(S_i(t, A))$ and $P'_i(t, A) = P'(S_i(t, A))$. Again, we drop A if A is understood.

3.3 CHARACTERIZING THE OPTIMAL SCHEDULE

In this section we characterize the optimal schedule for the objective of fractional flow plus energy. We start, in section 3.3.1, by giving a time-indexed convex programming formulation of the problem and then in section 3.3.2 we use the well known KKT conditions to derive properties that are necessary and sufficient for optimality. In section 3.3.3 we show that these properties imply that there is only a single optimal schedule. Finally, in section 3.3.4, we give a simple algorithm that uses the necessary and sufficient properties to decide whether or not a schedule is optimal in $O(n^2)$ time.

3.3.1 Convex Programming Formulation

We now give a convex programming formulation of the problem. Let T be some sufficiently large time bound such that all jobs finish by time T in the optimal schedule. Define the variable $S(t)$ as the speed of the schedule at time t and the variable $p_j(t)$ to be the work performed on job j at time $t \geq r_j$. The problem of minimizing a linear combination of fractional weighted flow plus energy for a set of jobs with release times and arbitrary work requirements can thus be expressed as the following convex program:

$$\min \sum_{j=1}^{n} \sum_{t \geq r_j}^{T} \frac{w_j p_j(t)}{p_j} (t - r_j) + \beta \sum_{t=1}^{T} P(S(t))$$

Subject to

$$\sum_{t=r_j}^{T} p_j(t) = p_j \qquad j \in [1, n] \quad [\text{dual } \alpha_j] \tag{3.1}$$

$$\sum_{j:r_j \leq t} p_j(t) = S(t) \qquad t \in [1, T] \quad [\text{dual } \delta(t)] \tag{3.2}$$

$$p_j(t) \geq 0 \qquad t \geq r_j, j \in [1, n] \quad [\text{dual } \gamma_j(t)] \tag{3.3}$$

The convexity of the program follows from the fact that, with the exception of $\sum P(S(t))$, the objective and constraints are linear functions of variables. $\sum P(S(t))$ is the non-negative sum of convex functions and thus is convex.

The objective has two terms, the right term is β times the energy used by the schedule. The left term is the sum over all jobs of the fractional weighted flow of the job. Constraint (3.1) ensures that every job is finished, constraint (3.2) ensures that the speed at each time is equal to the work performed at that time, and constraint (3.3) ensures that the work variables are non-negative. Constraints (3.2) and (3.3) ensure that the variables for speed are non-negative. To apply the KKT conditions in the next section, we associate the dual variables $\alpha_j, \delta(t)$, and $\gamma_j(t)$ with constraints, (3.1), (3.2), and (3.3) respectively.

3.3.2 KKT Conditions

The Karush-Kuhn-Tucker (KKT) conditions provide necessary and sufficient conditions to establish the optimality of feasible solutions to certain convex programs. We now describe the KKT conditions in general. Consider the following convex program,

$$\min f_0(x)$$

Subject to

$$\begin{aligned} f_i(x) &\leq 0 && i = 1, \ldots, n && (\lambda_i) \\ g_j(x) &= 0 && j = 1, \ldots, m && (\alpha_j) \end{aligned}$$

Assume that all f_i and g_j are differentiable and that there exists a feasible solution to this program. The variable γ_i is the dual (Lagrangian multiplier) associated with the function $f_i(x)$ and similarly for α_j with $g_j(x)$. Given that all constraints are differentiable linear functions and that the objective is differentiable and convex, the KKT conditions state that necessary and sufficient conditions for optimality are

$$f_i(x) \leq 0 \qquad i = 1, \ldots, n \tag{3.4}$$

$$\lambda_i \geq 0 \qquad i = 1, \ldots, n \tag{3.5}$$

$$\lambda_i f_i(x) = 0 \qquad i = 1, \ldots, n \tag{3.6}$$

$$g_j(x) = 0 \qquad j = 1, \ldots, m \tag{3.7}$$

$$\nabla f_0(x) + \sum_{i=1}^{n} \lambda_i \nabla f_i(x) + \sum_{j=1}^{m} \alpha_j \nabla g_j(x) = 0 \tag{3.8}$$

Here $\nabla f_i(x)$ and $\nabla g_j(x)$ are the gradients of $f_i(x)$ and $g_j(x)$ respectively. Condition 3.6 is called complementary slackness.

Before applying the KKT conditions, we define the *hypopower function* of job i in schedule A as,

$$Q_i(t, A) = q_i(A) - \frac{w_i}{\beta p_i}(t - r_i) \tag{3.9}$$

where $q_i(A)$ is any constant such that $Q_i(t, A)$ satisfies the condition that at all times t such that i is run in A, the hypopower at which i is run is $Q_i(t, A)$. If there is no such function (no $q_i(A)$) satisfying this condition, then i does not have an associated hypopower function in schedule A. Note that regardless of the speed to power function, $P(S)$, if the function $Q_i(t, A)$ exists (it may not), then the hypopower function of i is a linearly decreasing function of time with slope $\frac{-w_i}{\beta p_i}$. We refer to the constant $q_i(A)$ as i's *initial hypopower*. We drop A if the schedule is understood.

Because the hypopower function of i gives the hypopower of i, the hypopower function implies a speed function and a power function for i, specifically, $S_i(t, A) = P^*(Q_i(t, A))$ and $P_i(t, A) = P(P^*(Q_i(t, A)))$. Figure 3.3.2 gives a visual representation of the hypopower functions for the optimal schedule when $P(S) = S^3$ for an instance consisting of three jobs: with release times 0, 60, and 200, sizes 45, 35, and 25 and weights 0.1875, 0.25, and 0.125 respectively. Figure 3.3.2 shows the speed functions that correspond to the three hypopower functions of figure 3.3.2.

Lemma 6 states that a feasible schedule is optimal if and only if

- The hypopower of all jobs are defined by the hypopower function given in equation 3.9.
- At all times, the job that is run is the job whose hypopower function is largest at that time.
- If a job is run, it is run at the hypopower specified by the hypopower function.

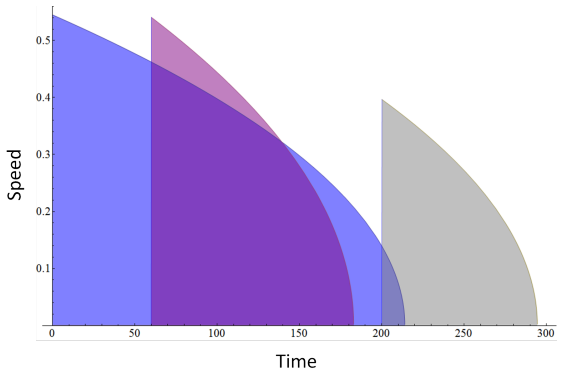

Figure 3: A three job instance viewed as speed functions.

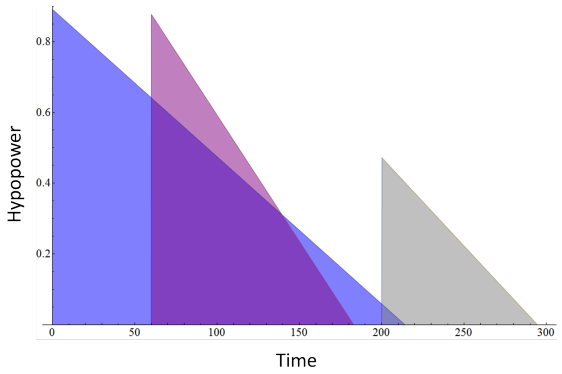

Figure 4: A three job instance viewed as hypopower functions.

Before we prove lemma 6, consider a couple of implications of these conditions.

First, an optimal schedule can be thought of as the upper envelope of the hypopower functions of all jobs. The set of times when the upper envelope is i's hypopower function are exactly the times during which i is run. Further, because the value of t that satisfies $Q_i(t) = Q_j(t)$, for any jobs i and j, is equal to the value of t that satisfies $S_i(t) = S_j(t)$, the upper envelope of the speed functions also defines an optimal schedule. The integral of the upper envelope of the speed functions equals the total work of all jobs and the integral of i's speed function, over the times when i's speed function defines the upper envelope, equals exactly i's total work. See figures 3.3.2 and 3.3.2 as examples of schedules defined by speed and hypopower functions respectively.

Second, because i's speed function is on the upper envelope of the speed functions if and only if i's hypopower function is on the upper envelope of the hypopower functions, the area under i's speed function, while i's speed function is on the upper envelope, can increase (or decrease) if and only if the area under i's hypopower function, while i's hypopower function is on the upper envelope, increases (or decreases). Thus, unless we need to calculate the specific work done on a job, it is generally easier to work with the hypopower functions of jobs rather than the speed functions of jobs because a job's hypopower functions is linear in time, while the speed function generally is not. Thus, unless explicitly stated otherwise, we will think of any schedule as n hypopower functions or equivalently as n initial hypopower values. Note that such a representation does not guarantee optimality, but does allow full description of any optimal schedule.

Lemma 6. *A primal feasible solution to the convex program is optimal if and only if, at all times t, for all jobs j, if $p_j(t) > 0$, then $P'(S(t)) = Q_j(t)$, and if $p_j(t) = 0$ then $P'(S(t)) \geq Q_j(t)$.*

Proof. We prove the lemma by showing that the KKT conditions are exactly the three conditions of the lemma: feasibility, the hypopower of A is $Q_j(t, A)$ whenever j is run, and $Q_j(t, A)$ is a lower bound on the hypopower of the schedule if j is not run.

First note that equations 3.4 and 3.7 of the KKT conditions are simply constraining all optimal solutions to be feasible, thus we need only show that the remaining two properties

are exactly equations 3.5, 3.6, and 3.8.

Because $q_j = P'(S(t^*)) + \frac{w_j}{\beta p_j}(t^* - r_j)$ for any time t^* such that j is run at t^*, it is sufficient to show the rest of the lemma for $P'(t) = P'(t^*) - \frac{w_j}{\beta p_j}(t - t^*)$ as this is equivalent to $Q_j(t)$. We start by computing the gradient of equation 3.8 of the KKT conditions. We then consider two cases for any job j: when j is running and when j is not running.

First consider equation 3.8 of the KKT conditions by first taking the partial derivative with respect to $S(t)$:

$$\beta P'(S(t)) - \delta(t) = 0 \text{ or equivalently } \delta(t) = \beta P'(t) \tag{3.10}$$

and $p_j(t)$:

$$\alpha_j - \gamma_j(t) + \delta(t) + (t - r_j)\frac{w_j}{p_j} = 0$$

or equivalently,

$$\alpha_j = \gamma_j(t) - \delta(t) - (t - r_j)\frac{w_j}{p_j} \tag{3.11}$$

We can then plug equation 3.10 into equation 3.11 to get

$$\alpha_j = \gamma_j(t) - \beta P'(t) - (t - r_j)\frac{w_j}{p_j} \tag{3.12}$$

Thus, whatever the value of α_j, it is constant for job j at all times $t \geq r_j$.

For the first case, consider any job j, and time, t^*, when j is run. Call the speed of the schedule at t^*, $S(t^*)$. By equation 3.12,

$$\alpha_j = \gamma_j(t^*) - \beta P'(t^*) - (t^* - r_j)\frac{w_j}{p_j} \tag{3.13}$$

Now consider any other time t during which j is run. Again by equation 3.12 we have,

$$\alpha_j = \gamma_j(t) - \beta P'(t) - (t - r_j)\frac{w_j}{p_j} \tag{3.14}$$

Equating equations 3.13 and 3.14 for α_j and solving for $\beta P'(t)$ gives us,

$$\beta P'(t) = -\gamma_j(t^*) + \gamma_j(t) + \beta P'(t^*) - (t - t^*)\frac{w_j}{p_j} \tag{3.15}$$

However, applying complementary slackness (3.6) to constraint 3.3 of our convex program, we get that $\gamma_j(t)(-p_j(t)) = 0$ and $\gamma_j(t^*)(-p_j(t^*)) = 0$. However, because we know that at both t^* and t, job j is run, both $p_j(t) > 0$ and $p_j(t^*) > 0$, thus it must be that $\gamma_j(t) = 0$ and $\gamma_j(t^*) = 0$, thus equation 3.15 becomes,

$$\beta P'(t) = \beta P'(t^*) - (t - t^*)\frac{w_j}{p_j}$$

or equivalently,

$$P'(t) = P'(t^*) - \frac{w_j}{\beta p_j}(t - t^*)$$

Thus we have the second condition of our lemma. Lastly, note that equation 3.5 of the KKT conditions is satisfied by $\gamma_j(t) = 0$, and it must be that $\gamma_j(t) = 0$ in order to satisfy complementary slackness (3.6) in the case that job j is run at time t.

For the second case, consider any time, t', such that job j is not run. We apply equation 3.12 to get

$$\alpha_j = \gamma_j(t') - \beta P'(t') - (t' - r_j)\frac{w_j}{p_j}$$

Because α_j is constant whether j is run or not, we then set this equal to equation 3.13 and solve for $P'(t')$ to get,

$$\begin{aligned} P'(t') &= -\frac{\gamma_j(t^*)}{\beta} + \frac{\gamma_j(t')}{\beta} + P'(t^*) - (t' - t^*)\frac{w_j}{\beta p_j} \\ &= \frac{\gamma_j(t')}{\beta} + P'(t^*) - (t' - t^*)\frac{w_j}{\beta p_j} \\ &\geq P'(t^*) - (t' - t^*)\frac{w_j}{\beta p_j} \end{aligned}$$

The equality follows from the fact that $\gamma_j(t^*) = 0$ and the inequality follows by the fact that $\beta \geq 0$ and by equation 3.5 of the KKT conditions which requires that $\gamma_j(t') \geq 0$. Thus we have the third condition of our lemma. Lastly, note that because $p_j(t') = 0$ when job j is not run at t', complementary slackness (3.6) is thus always satisfied in this case. □

The job selection policy highest density first (HDF), schedules, at time t, the unfinished job i, with the largest density, which is defined to be w_i/p_i. By using a standard exchange argument one can show that HDF is the optimal job selection policy for weighted fractional delay. Thus we expect that any feasible schedule meeting the conditions of Lemma 6 schedules jobs in HDF order. To see why this is indeed true, consider that the slope of $Q_i(t)$ is dependent only on β and i's density. Thus when $Q_i(t)$ and $Q_j(t)$ intersect, the less dense job will have a larger hypopower at all times after the intersection. Lemma 6 then implies that, in the optimal schedule, the denser of i and j must have been completed prior to the intersection of $Q_i(t)$ and $Q_j(t)$. Because this holds for all jobs, if $Q_i(t)$ is on the upper envelope at time t, then because $Q_i(t) \leq Q_j(t)$ for all j released by t, all released jobs with density larger than i's density must have been completed by t. However, this is the definition of HDF, thus the conditions of Lemma 6 imply HDF job selection.

3.3.3 The Optimal Schedule is Unique

In this section, we show that the optimal schedule is unique. We do this by examining the upper envelope of the hypopower functions for two purportedly optimal schedules, A and B, and consider the set of jobs H such that i is in H if the initial hypopower of i in A is strictly smaller than the initial hypopower of i in B. We show that area under the hypopower upper envelope, over all times a schedule is running any job in H, is larger for schedule B than schedule A. This implies the same for the speed functions, that is, the total work done on jobs in H is different in A and B, a contradiction to both schedules being optimal.

Lemma 7. *The optimal schedule is unique.*

Proof. We will prove this lemma by contradiction. Specifically, assume there are two optimal schedules, A, and B. We will convert A into B by changing the jobs one at a time. If the set H consists of all jobs i such that $q_i(A) < q_i(B)$, then we show that every time we change a job in H, the total work done on jobs in H either goes up or stays the same. And every time we change a job not in H the total work done on jobs in H either goes up or stays the same. Finally, we show that the work done on jobs in H goes up at least once, thus A does less work on jobs in H than B, a contradiction to both schedules being optimal. Instead

of looking at work directly we will look at area under the hypopower curves. Proving this quantity increases implies the same for the area under the corresponding speed functions, thus completing the contradiction.

Assume A and B are each represented by n q_i values, thus schedule A can be thought of as $Q(t, A) = max_i \{Q_i(t, A)\}$ and B as $Q(t, B) = max_i \{Q_i(t, B)\}$. For any set of jobs S, define $Q(t, S \in A)$ as 0 if $max_i \{Q_i(t, A)\} > max_{i \in S} \{Q_i(t, A)\}$ and $max_{i \in S} \{Q_i(t, A)\}$ otherwise. We can similarly define $Q(t, S \in B)$. In other words, these functions are the subset of the upper envelope where some job in S is the running job. Without loss of generality, assume A has at least one job, j, such that $q_j(A) < q_j(B)$. Call the set H, all jobs, i, such that $q_i(A) < q_i(B)$ and the set L, all jobs, i, such that $q_i(A) > q_i(B)$. We can convert A into B by setting, one at a time, $q_k(A)$ to $q_k(B)$, for each job k. Consider what happens when we do this for job arbitrary job k:

If $k \in H$, then $q_k(A) < q_k(B)$. Thus $Q_k(t, A)$ increases at all times t, but $Q_j(t, A)$, for all $j \neq k$ and time t, does not decrease. These facts imply that the area under $Q(t, H \in A)$ either increases or stays the same. Further, if prior to increasing $q_k(A)$, it is the case that $Q_k(t, A) = Q(t, H \in A)$ for any t, then the area under $Q(t, H \in A)$ strictly increases.

If $k \in L$, then $q_k(A) > q_k(B)$. Thus $Q_k(t, A)$ decreases at all times t, but $Q_j(t, A)$, for all $j \neq k$ and time t, does not decrease. These facts also imply that the area under $Q(t, H \in A)$ either increases or stays the same.

We still need at least a single increase in the area under $Q(t, H \in A)$. However, recall that we are guaranteed to have some job j such that such that $q_j(A) < q_j(B)$. Thus if we convert A to B by starting with j, it must be that $Q_j(t, A) = Q(t, H \in A)$ for at least one time t, else A does no work on j, a contradiction to the optimality of A. □

3.3.4 Checking a Schedule for Optimality

We conclude section 3.3 by giving an algorithm to check the optimality of a schedule in time $O(n^2)$. The algorithm takes as input the initial hypopower for each job j. If the input schedule is not in this form, then $q_i = P_i'(t^*) + (w_i/\beta p_i)(t^* - r_i)$ for any time t^* when j is run in the input schedule. If the resulting hypopower functions are not optimal then the input

schedule is not optimal. If the resulting hypopower functions are optimal, then determining if the input schedule is optimal reduces to the problem of deciding if the input schedule is identical to the schedule produced by the upper envelope of the resulting hypopower functions.

Because the hypopower functions are linear, when two hypopower functions intersect, the function defined by a job of lower density will be strictly larger at all times after the intersection. Thus any hypopower function is involved in at most 1 crossing on the upper envelope with a lower density job, specifically the earliest crossing with a lower density job. Thus there are most n such crossings, and it is not too hard to see that for each hypopower function, $Q_i(t)$, we can find, in linear time, the earliest time, if it exists, that $Q_i(t)$ crosses some $Q_j(t)$ such that $w_i/p_i > w_j/p_j$ and the crossing time is at least $max\{r_i, r_j\}$. Thus one can, in $O(n^2)$ time, compute the upper envelope of the hypopower functions.

3.4 APPLYING THE HOMOTOPIC APPROACH

The main idea of the homotopic approach is to start with a schedule we can easily compute for some β' and slowly change β', calculating the new optimal schedule each time we do so, until $\beta' = \beta$. This seemingly requires that the optimal schedules for infinitesimally different β's must be closely related, so to find the new optimal schedule, when we change β' to $\beta' + \epsilon$, we only have to examine schedules that are close to the previously computed optimal schedule for β'. In section 3.4.1 we discuss how to easily find an initial optimal schedule that can be used as the starting point for a homotopic algorithm. In section 3.4.2 we prove that the initial hypopowers change continuously as a function of β. (Although, perhaps somewhat counter-intuitively, the initial hypopowers are not monotone in β.) This then allows us to obtain an efficient homotopic algorithm for computing the optimal energy trade-off schedule for a small number of jobs by simply searching over schedules with nearly identical initial hypopowers.

3.4.1 Finding an Initial Optimal Schedule

If the processor has a maximum speed, then initially $\beta = 0$, and the optimal schedule always runs at the maximum speed if there are unfinished jobs, and uses HDF to determine which job to run. If the processor does not have a maximum speed, we choose β small enough such that every job is completed before any other jobs are released. For each job j, we can calculate a β_j such that it completes before any other job is released, and then take β to be the minimum over all β_j.

3.4.2 The Optimal Schedule Changes Continuously

In this section we show that the initial hypopowers are a continuous function of β. To this end, we first define the initial hypopower of job j, as a function of β, as $q_j(\beta)$ and likewise the hypopower function for j, as a function of β, as $Q_j(t, \beta)$.

We show in Lemma 8 that the optimal schedule changes continuously as a function of β. By Lemma 6, the optimal schedule can be described as set of n initial hypopowers, thus we show Lemma 8 by showing that these initial hypopowers are continuous functions of β. We do this by showing that if there is some non-empty set of jobs whose initial hypopowers are increasing discontinuously at β, then for some small increase in β, the total work done by the optimal schedule on this set of jobs increases. However, this is a contradiction to optimality. If the initial hypopowers are decreasing continuously, then the same method can be used for a small decrease in β.

Lemma 8. *The initial hypopowers are a continuous function of β.*

Proof. We show the lemma by showing that, in the optimal schedule, for all jobs j, the value $q_j(\beta)$ is a continuous function of β. That is, assume that there is at least one $q_j(\beta)$ value that is not continuous in β. More precisely, $q_j(\beta)$ is *discontinuously increasing* at $\beta > 0$ if there exists constants $c_1, c_2 > 0$ such that for all $\epsilon \in (0, c_2)$, $q_j(\beta + \epsilon) \geq q_j(\beta) + c_1$. Likewise, $q_j(\beta)$ is *discontinuously decreasing* at $\beta > 0$ if there exists constants $c_1, c_2 > 0$ such that for all $\epsilon \in (0, c_2)$, $q_j(\beta - \epsilon) \geq q_j(\beta) + c_1$.

We start by assuming the following claim:

Claim 1. *For any job j with discontinuously increasing $q_j(\beta)$, there exists some $\epsilon' \in (0, c_2)$ such that the following two properties hold:*

1. *For all times t, $Q_j(t, \beta') > Q_j(t, \beta)$ for all $\beta' \in (\beta, \beta + \epsilon']$*
2. *For all jobs i such that $q_i(\beta)$ is not discontinuously increasing at β, define t_c as the solution of $Q_j(t_c, \beta) = Q_i(t_c, \beta)$ and t'_c as the solution of $Q_j(t'_c, \beta') = Q_i(t'_c, \beta')$, then for all $\beta' \in (\beta, \beta + \epsilon']$:*
   - *If $w_i/p_i > w_j/p_j$ then $t_c > t'_c$ else*
   - *If $w_i/p_i < w_j/p_j$ then $t_c < t'_c$*

Likewise if $q_j(\beta)$ is discontinuously decreasing, then the same facts hold except $\beta' \in [\beta - \epsilon', \beta)$.

The proof of the lemma is now the same as for Lemma 7: we convert from one schedule into another by changing jobs one at a time and show that there is some set of jobs such that the total work increases. For the sake of contradiction, assume that there is at least one job such that the job's $q(\beta)$ is discontinuous at some value of β. There are two cases, if $q(\beta)$ is discontinuously increasing and if $q(\beta)$ is discontinuously decreasing.

If there is at least one discontinuously increasing $q(\beta)$ function, consider the smallest $\beta = \beta_1$ such that there are some set of jobs, H with $q(\beta)$ functions that are discontinuously increasing at β_1. By Claim 1, there exists some $\epsilon' > 0$, such that the properties of Claim 1 hold for all jobs in H. We now convert the optimal schedule at β_1 to the optimal schedule at any $\beta' \in (\beta_1, \beta_1 + \epsilon']$ following the proof of Lemma 7 with the main difference being that for some jobs $i \notin H$, it may be that q_i increases, however Claim 1 ensures that changing them does not decrease the work done on any job in H.

If there is at least one discontinuously decreasing $q(\beta)$ function, we follow the same method except we start from the largest such β_1.

All that remains is to show Claim 1.

First note that if we find an ϵ'_1 satisfying the first property and an ϵ'_2 satisfying the second property, then $\epsilon' = min\{\epsilon'_1, \epsilon'_2\}$ will satisfy both properties. Thus we can find an ϵ' value separately for each property.

We start by showing that if $q_j(\beta)$ is discontinuously increasing at β, the first property holds. In other words, we want to find an ϵ' such that the following holds for any $\beta' \in$

$(\beta, \beta + \epsilon']$:

$$q_j(\beta) - \frac{w_j}{p_j\beta}(t - r_j) < q_j(\beta') - \frac{w_j}{p_j\beta'}(t - r_j)$$

As ϵ' is constrained to be less than c_2 and by definition of the discontinuity of $q_j(\beta)$, we have that,

$$q_j(\beta) + c_1 - \frac{w_j}{p_j\beta'}(t - r_j) < q_j(\beta') - \frac{w_j}{p_j\beta'}(t - r_j)$$

Thus it is sufficient to show

$$q_j(\beta) - \frac{w_j}{p_j\beta}(t - r_j) < q_j(\beta) + c_1 - \frac{w_j}{p_j\beta'}(t - r_j)$$

Or equivalently,

$$-\frac{w_j}{p_j\beta}(t - r_j) < c_1 - \frac{w_j}{p_j\beta'}(t - r_j) \tag{3.16}$$

However, this is clearly true for all $\beta' > \beta$ as $c_1 > 0$, thus any $\epsilon' \in (0, c_2)$ will satisfy the inequality. If instead $q_j(\beta)$ discontinuously increasing, and we require $\beta' \in [\beta - \epsilon', \beta)$, we can re-arrange inequality 3.16 to get

$$\frac{w_j}{p_j}(t - r_j)\left(\frac{1}{\beta'} - \frac{1}{\beta}\right) < c_1 \tag{3.17}$$

Because $(w_j/p_j)(t - r_j)\,(1/\beta' - 1/\beta)$ is a decreasing function of β', if we can find a single $\beta' = \beta - \epsilon'$ such that 3.17 holds then we are done. However, consider that for $\epsilon' = 0$,

$$\frac{w_j}{p_j}(t - r_j)\left(\frac{1}{\beta - \epsilon'} - \frac{1}{\beta}\right) < c_1$$

holds. Thus, because $(w_j/p_j)(t - r_j)\,(1/(\beta - \epsilon') - 1/\beta)$ is continuous in ϵ' and $c_1 > 0$, there must be some $\epsilon' > 0$ for which this holds.

Now we show that if $q_j(\beta)$ is discontinuously increasing at β, then the second property holds. First we give the explicit definition of t_c and t'_c:

$$t_c = \frac{(q_j(\beta) - q_i(\beta))\beta + \frac{w_j r_j}{p_j} - \frac{w_i r_i}{p_i}}{\frac{w_j}{p_j} - \frac{w_i}{p_i}} \tag{3.18}$$

Likewise, solving for t'_c gives

$$t'_c = \frac{(q_j(\beta') - q_i(\beta'))\beta' + \frac{w_j r_j}{p_j} - \frac{w_i r_i}{p_i}}{\frac{w_j}{p_j} - \frac{w_i}{p_i}} \tag{3.19}$$

Now we would like to show that $t_c > t'_c$ if $w_i/p_i > w_j/p_j$ and $t_c < t'_c$ if $w_i/p_i < w_j/p_j$. Equivalently, $t'_c - t_c < 0$ and $t'_c - t_c > 0$ respectively. Solving directly for $t'_c - t_c$ using equations 3.18 and 3.19 we get

$$t'_c - t_c = \frac{(q_j(\beta') - q_i(\beta'))\beta' - (q_j(\beta) - q_i(\beta))\beta}{\frac{w_j}{p_j} - \frac{w_i}{p_i}}$$

Note that if $w_i/p_i > w_j/p_j$, then $w_j/p_j - w_i/p_i < 0$ and if $w_i/p_i < w_j/p_j$ then $w_j/p_j - w_i/p_i > 0$, thus for both cases it is sufficient to show that

$$(q_j(\beta') - q_i(\beta'))\beta' - (q_j(\beta) - q_i(\beta))\beta > 0$$

By definition of job j and the valid range of β', $q_j(\beta') \geq q_j(\beta) + c_1$, thus giving us

$$\begin{aligned}(q_j(\beta') - q_i(\beta'))\beta' - (q_j(\beta) - q_i(\beta))\beta \geq \\ (q_j(\beta) + c_1 - q_i(\beta'))\beta' - (q_j(\beta) - q_i(\beta))\beta\end{aligned}$$

Or equivalently,

$$\begin{aligned}(q_j(\beta') - q_i(\beta'))\beta' - (q_j(\beta) - q_i(\beta))\beta \geq \\ c_1\beta' + q_j(\beta)(\beta' - \beta) + q_i(\beta)\beta - q_i(\beta')\beta'\end{aligned} \tag{3.20}$$

In the case that $q_j(\beta)$ is discontinuously increasing, we have that $\beta' > \beta$, which implies that $q_j(\beta)(\beta' - \beta) > 0$ and $c_1\beta' > c_1\beta$. Thus it is sufficient to show

$$c_1\beta + q_i(\beta)\beta - q_i(\beta')\beta' > 0 \tag{3.21}$$

If $q_i(\beta)$ is continuous at β, the limit of $q_i(\beta)\beta - q_i(\beta')\beta'$ as β' goes to β is 0. If $q_i(\beta)\beta - q_i(\beta')\beta'$ goes to 0 from larger than 0, then inequality 3.21 holds. Because $c_1\beta$ is strictly larger than 0, if $q_i(\beta)\beta - q_i(\beta')\beta'$ goes to 0 from less than 0, then we can make $q_i(\beta)\beta - q_i(\beta')\beta'$ arbitrarily close to 0, in a continuous manner, as we decrease β'. Thus, there must be some $\epsilon' > 0$ such that $-(q_i(\beta)\beta - q_i(\beta + \epsilon'')) < c_1\beta$ for all $\epsilon'' < \epsilon'$.

If however $q_i(\beta)$ is discontinuously decreasing at β, then by the first property, there is some small ϵ_1 such that for $\beta_1 \in (\beta, \beta + \epsilon_1]$, $Q_i(t, \beta_1) < Q_i(t, \beta)$ for all t. This implies that $q_i(\beta)\beta - q_i(\beta')\beta' > 0$ for $\beta' \in (\beta, \beta + \epsilon_1]$, thus inequality 3.21 holds for any $\epsilon' < \epsilon_1$.

Finally consider the case that $q_i(\beta)$ is discontinuously decreasing at β and thus we want $\beta' \in [\beta - \epsilon, \beta)$. The only difference is that $c_1\beta' \geq c_1(\beta - \epsilon') = c_1\beta - \epsilon' c_1$ and $q_j(\beta)(\beta' - \beta) \geq -q_j(\beta)\epsilon'$. Applying these to equation 3.20, we need to show the following for all β',

$$c_1\beta + q_i(\beta)\beta - q_i(\beta')\beta' - \epsilon' c_1 - q_j(\beta)\epsilon' > 0 \tag{3.22}$$

However, consider that both $-\epsilon' c_1$ and $-q_j(\beta)\epsilon'$ are continuous and go to 0 as ϵ' goes to 0. In the case that $q_i(\beta)$ is continuous, $q_i(\beta)\beta - q_i(\beta')\beta' - \epsilon' c_1 - q_j(\beta)\epsilon'$ is thus continuous, has a limit of 0 at $\epsilon' = 0$, and we can therefore use the same reasoning as in the previous continuous case. If $q_i(\beta)$ is discontinuous, it must be discontinuously increasing in β or in other words, discontinuously decreasing as we lower β. Thus there exists some ϵ_1 such that for all $\beta' \in [\beta - \epsilon_1, \beta)$, $q_i(\beta)\beta - q_i(\beta')\beta' \geq c_3 > 0$, but this just makes the left hand side of inequality 3.22 larger than if $q_i(\beta)$ was continuous. □

3.5 PRODUCT OBJECTIVES

In this section we consider objectives of the form $\mathcal{Q}^\sigma \times \mathcal{E}$, where $\mathcal{Q}$ is a scheduling objective, $\mathcal{E}$ is energy, and σ is a parameter representing the relative importance of the scheduling objective versus energy. In Lemma 9, we show that (perhaps counter-intuitively) this product objective is not particularly interesting from a theoretical perspective if the static power is zero. We do this by showing that if the power function is of the form $P(s) = s^\alpha$, then the optimal solution is to either always go as fast as possible or to always go as slow as possible.

When the scheduling objective $\mathcal{Q}$ is integer flow, we are unable to characterize the optimal scheduling for the objective $\mathcal{Q}^\sigma \times \mathcal{E}$ for the same reasons that we are unable to characterize the optimal schedules for the additive objective with integral flow. We therefore only consider fractional flow. The next issue that arises is during what time period does one count the static power's contribution to energy. Perhaps the most natural assumption might be to only

count static power when the processor is running jobs. But this has various mathematical issues, such as it then becomes difficult to write a reasonable mathematical program. Thus we assume that there is a specified time period T, add a constraint that all jobs much be finished by time T, and assume that static power accumulates during exactly the period T. So the energy arising from static power is $P(0)T$. We characterize the optimal schedule in Lemma 10. In Lemma 11 and Lemma 12 we show that, for a fixed instance, the schedules that are optimal for the product objective $\mathcal{Q}^\sigma \times \mathcal{E}$ (for any σ) are a (generally strict) subset of the schedules that are optimal for the sum objective $\mathcal{Q} + \beta\mathcal{E}$ (for any β). We then show in Lemma 13 that it is unlikely that one will be able to compute an optimal schedule for $\mathcal{Q}^\sigma \times \mathcal{E}$ using a homotopic approach as the optimal schedule may be a discontinuous function of σ.

Lemma 9. *Assume that the power function is $P(s) = s^\alpha$ and the scheduling objective $\mathcal{Q}$ is either fractional flow or integral flow. Then for every instance, the optimal schedule for the objective $\mathcal{Q}^\sigma \times \mathcal{E}$ either always runs the processor as slowly as possible, or always runs the processor as fast as possible.*

Proof. We give the proof for integral flow; the proof for fractional flow is similar. Consider an instance with a single job with work 1. When running at constant speed s, the time to finish the job is $1/s$ and the energy used is $s^\alpha/s = s^{\alpha-1}$, which yields an objective value of $\frac{s^{\alpha-1}}{s^\sigma} = s^{\alpha-\sigma-1}$. Thus, if $\alpha - \sigma - 1 > 0$, the objective is minimized at 0 by running as slow as possible, while if $\alpha - \sigma - 1 < 0$, the objective is minimized again at 0 by running as fast as possible. Finally, if $\alpha - \sigma - 1 = 0$, any schedule that runs the job at constant speed is optimal so running either as fast or as slow as possible is optimal. In any unweighted, unit work n job instance, if $\alpha - \sigma - 1 \leq 0$, the schedule that runs jobs as fast as possible is still optimal, and if $\alpha - \sigma - 1 > 0$ the schedule that runs jobs as slow as possible is also still optimal, since the increased flow at every time only increases the objective by a factor of $O(n^{2\sigma})$. It is straightforward to see that this to extends when jobs have weights and arbitrary work requirements since in this case the objective would, at worst, be multiplied by an additional value dependent on the input, but not the schedule. □

The problem of minimizing $\mathcal{Q}^\sigma \times \mathcal{E}$ over a time period T can be expressed as the following mathematical program:

$$\min \left(\sum_{j=1}^{n} \sum_{t \geq r_j}^{T} \frac{w_j p_j(t)}{p_j} (t - r_j) \right)^{\sigma} \left(\sum_{t=1}^{T} P(S(t)) \right)$$

Subject to:

$$\sum_{t=r_j}^{T} p_j(t) = p_j \qquad j \in [1, n] \quad [\text{dual } \alpha_j] \tag{3.23}$$

$$\sum_{j:r_j \leq t} p_j(t) = S(t) \qquad t \in [1, T] \quad [\text{dual } \delta(t)] \tag{3.24}$$

$$p_j(t) \geq 0 \qquad t \geq r_j, j \in [1, n] \quad [\text{dual } \gamma_j(t)] \tag{3.25}$$

Note that these constraints are the same as (3.1), (3.2), and (3.3) from section 3.3.1. Unlike there, however, this mathematical program is not convex, and so the KKT conditions (section 3.3.2) provide only necessary conditions for optimality.

Before applying the KKT conditions, we define a new hypopower function of job i in schedule A as,

$$\widetilde{Q}_i(t, A) = \tilde{q}_i(A) - \frac{\mathcal{E}(\mathcal{A})}{\mathcal{Q}(\mathcal{A})} \frac{\sigma w_i}{p_i} (t - r_i) \tag{3.26}$$

where $\mathcal{E}(\mathcal{A})$ is the total energy used by schedule A, $\mathcal{Q}(\mathcal{A})$ is the total fractional flow incurred by schedule A, and $\tilde{q}_i(A)$ is any constant such that $\widetilde{Q}_i(t, A)$ satisfies the condition that at all times t such that i is run in A, the hypopower at which i is run is $\widetilde{Q}_i(t, A)$. Note that (3.26) is very similar to the hypopower function defined by equation (3.9), and the same qualifications apply. The only difference is that the slope is now $\frac{\mathcal{E}(\mathcal{A})}{\mathcal{Q}(\mathcal{A})} \frac{\sigma w_i}{p_i}$, which is a function of σ as well as the total energy and total flow used by the entire schedule.

Lemma 10 states that if a feasible schedule is optimal, the following must be true:

- The hypopower of all jobs are defined by the hypopower function given in equation 3.26.
- At all times, the job that is run is the job whose hypopower function is largest at that time.
- If a job is run, it is run at the hypopower specified by the hypopower function.

Lemma 10. *If a solution to the mathematical program is optimal then at all times t, for all jobs j, if $p_j(t) > 0$, then $P'(S(t)) = \widetilde{Q}_j(t)$, and if $p_j(t) = 0$ then $P'(S(t)) \geq \widetilde{Q}_j(t)$.*

Proof. This proof is almost identical to the proof of lemma 6, so only the differences are highlighted, with the main difference being the difference in objective functions. In regards to the different objective, first note that minimizing the objective $\mathcal{Q}^\sigma \times \mathcal{E}$ is equivalent to minimizing $\sigma \log(\mathcal{Q}) + \log(\mathcal{E})$. Thus, when we take the partial derivative with respect to $S(t)$ we obtain

$$\frac{P'(S(t))}{\sum_{t'=0}^{T} P(S(t'))} - \delta(t) = 0$$

or equivalently,

$$\delta(t) = \frac{P'(S(t))}{\mathcal{E}} \tag{3.27}$$

and $p_j(t)$:

$$\alpha_j - \gamma_j(t) + \delta(t) + \frac{\sigma\,(t - r_j)\,\frac{w_j}{p_j}}{\sum_{j'=1}^{n} \sum_{t' \geq r_{j'}}^{T} \frac{w_{j'} p_{j'}(t')}{p'_j}\,(t' - r_{j'})} = 0$$

or equivalently,

$$\alpha_j = \gamma_j(t) - \delta(t) - \frac{\sigma\,(t - r_j)\,w_j}{\mathcal{Q}\,p_j} \tag{3.28}$$

Observe that (3.27) and (3.28) are almost identical to (3.10) and (3.11) from lemma 6, the only differences being the coefficients on some of the terms. The rest of the proof of this lemma follows exactly as the proof of lemma 6, only requiring the additional observation at one point that $\mathcal{E} \geq 0$. □

Lemma 11. *For any instance, the set of all optimal schedules for the product objective (for any σ) is a subset of the set of all optimal schedules for the sum objective (for any β).*

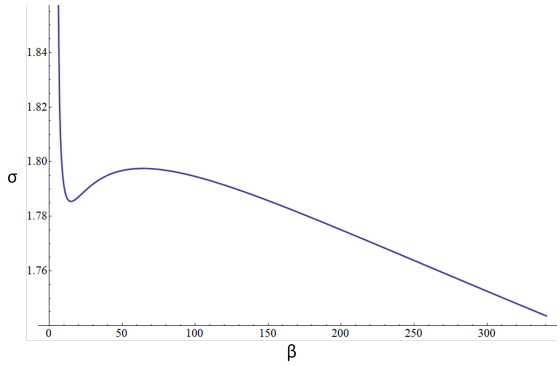

Figure 5: β vs σ for schedules satisfying the KKT conditions in a two job instance.

Proof. Fix an arbitrary σ. In the optimal schedule for that σ, if it exists, each job has a linearly decreasing hypopower function with slope $\sigma\frac{\mathcal{E}}{\mathcal{Q}}$ times its density. If we set $1/\beta = \sigma\frac{\mathcal{E}}{\mathcal{Q}}$, then we have a set of identical hypopower functions for the sum objective for that β. Thus if the schedule is optimal for that σ in the product objective, it must also be optimal for β in the sum objective. Thus the optimal schedule for any σ in the product objective has a corresponding optimal schedule for some β in the sum objective. □

Lemma 12. *There are instances where there are schedules that are optimal for the sum objective but are not optimal for the product objective for any σ.*

Proof. Consider a two job instance where the jobs have release time 0 and 60, work 11 and 7, and weight 0.15 and 1.5, respectively. The power function is $P(s) = s^3 + .01/T$, where T is sufficiently big so that finishing before T is not a tight constraint. Thus the total static energy used is 0.01. As an example, see Figure 3.5 with $\sigma = 1.79$ and Figure 3.5. For each β, Figure 3.5 shows the value of σ for which the optimal sum schedule is a locally optimum schedule for the produce objective. Simple calculations show that different local optimums have different objective values in the product objective. □

Lemma 13. *The optimal schedule for the product objective is not necessarily a continuous function of σ.*

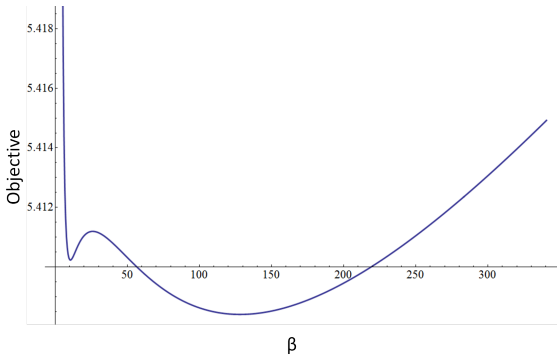

Figure 6: β vs (the log of) $\mathcal{Q}^\sigma \times \mathcal{E}$ for $\sigma = 1.79$ in the same instance as Figure 3.5. The β values that give $\sigma = 1.79$ in in Figure 3.5 are approximately 10, 26, and 128. Note how they yield locally optimal solutions here.

Proof. To see this, consider the example in Figure 3.5. Here, for all σ less than $\sigma_1 \approx 1.785$ or greater than $\sigma_2 \approx 1.80$ there is only one β that maps to it, but there are multiple β that map to all $\sigma \in [\sigma_1, \sigma_2]$. Further, in this case, for any σ, only one of the up to three β's that provide KKT condition satisfying schedules for that σ minimize the objective. This implies that the function mapping a σ to the β that shares its optimal schedule is not continuous since it is an inverse function of a continuous function that is not one-to-one. This further implies the optimal schedule does not change continuously from σ_1 to σ_2, since a discontinuous increase (or decrease) in β implies a discontinuous decrease (or increase) in the slopes of all hypopower functions, so if the initial hypopowers did not also change discontinuously, the resulting schedule after the discontinuity would do too little (or too much) work, and thus not be feasible (or optimal). □

3.6 OPEN QUESTIONS

Our investigations have revealed some natural open questions, which we list below. We believe that it is quite plausible that the introduction of these problems into the literature

might be the most lasting contribution of this paper.

Question 1: *Given the configuration of the optimal energy trade-off schedule (when the quality of service measure is weighted fractional flow), can the optimal schedule be computed in polynomial time?* In this context, a configuration is the sequence of jobs run on the processor as well as whether a job completes before, at, or after the release of the next job to be run (which is not necessarily the next job that was released). This seems to be the natural definition of a configuration, as the configuration and β uniquely defines a set of equations for which the initial hypopowers of the optimal schedule for that configuration are the sole (real) solution. [31] was able to use binary search to solve these equations. We were unable to compute the optimal schedule even knowing the optimal configuration. The main difficulty is that even when the power function is s^α the configuration equations yield a series of α-degree polynomial equations (in the variables hypopower, or speed, or interval length). We know of no technique to acquire a general algebraic solution to these equations. Another natural alternative approach is to create a mathematical program from the equations, and solve them using some standard optimization algorithm. However, any formulation we examined had equations relating to work that were not convex (again, in variables hypopower, or speed, or interval length). More precisely, let $p_j(\vec{x})$ be the amount of work done on job j for some variable assignment $\vec{x}$. Then for variable assignments $\vec{x}$ and $\vec{y}$, where $p_j(\vec{x}) = p_j(\vec{y}) = p_j$, there exist some situations where $p_j((\vec{x}+\vec{y})/2) > p_j$ and other situations where $p_j((\vec{x}+\vec{y})/2) < p_j$. Thus one can not seemingly use convex optimization.

Question 2: *Is there an efficient algorithm to compute the optimal energy trade-off schedule (when the quality of service measure is weighted fractional flow) for $\beta + \epsilon$ given the optimal trade-off schedule for β?* As we know how to detect when the optimal configuration changes and how to find the new optimal configuration, this is closely related to the previous open question.

Question 3: *As a function of the number of jobs n, how many times can the optimal configuration change as β changes?* [31] shows that for unit jobs and when the quality of service measure is total delay, the number of configuration changes is $O(n^2)$. We would be quite surprised if the number of configurations changes in our setting was not also polynomially

bounded, although we do not know how to prove any upper bound that is a function of n. The problem is because the initial hypopowers are not monotone in β, we do not even know how to show that a particular configuration will be optimal for a contiguous collection of β's.

Question 4: *Is there a polynomial time algorithm to verify the optimality of schedule for the objective of total (integer) delay plus energy?* Using insights from [31], one can design a polynomial time algorithm that given a schedule S and a configuration C, can verify the optimality of S among schedules in configuration C. But we do not know how to verify that a schedule is in the optimal configuration.

Question 5: *For what quality of service measures can homotopic optimization be used to compute optimal energy trade-off schedules?* It is not completely implausible that one can characterize the natural quality of service measures where this is possible.

4.0 SPEED SCALING TO MANAGE TEMPERATURE

In this section, we consider the speed scaling problem where the quality of service objective is deadline feasibility and the power objective is temperature. In the case of batched jobs, we give a simple algorithm to compute the optimal schedule. For general instances, we give a new online algorithm, and obtain an upper bound on the competitive ratio of this algorithm that is an order of magnitude better than the best previously known bound upper bound on the competitive ratio for this problem. We also show that by using our analysis, we can improve the temperature analysis of the energy optimal offline algorithm, YDS.

4.1 RELATED WORK

[33] showed that there is a greedy offline algorithm YDS to compute the energy optimal schedule. A naive YDS implementation runs in time $O(n^3)$, which is improved in [28] to $O(n^2 \log n)$. [33] suggested two online algorithms OA and AVR. OA runs at the optimal speed assuming no more jobs arrive in the future (or alternately plans to run in the future according to the YDS schedule). AVR runs each job at an even rate between its release time and deadline. In a complicated analysis, [33] showed that AVR is at most $2^{\alpha-1}\alpha^{\alpha}$-competitive with respect to energy. A simpler competitive analysis of AVR, with the same bound, as well as a nearly matching lower bound on the competitive ratio for AVR can be found in [6]. [10] shows that OA is α^{α}-competitive with respect to energy. [10] showed how potential functions can be used to give relatively simple analysis of the energy used by an online algorithm. [9] introduces an online algorithm qOA, which runs at a constant factor q faster than OA, and shows that qOA is at most $4^{\alpha}/(2\sqrt{e\alpha})$-competitive with respect to

energy. When the cube root rule holds, qOA has the best known competitive ratio with respect to energy, namely 6.7. [9] also gives the best known general lower bound on the competitive ratio, for energy, of deterministic algorithms, namely $e^{\alpha-1}/\alpha$.

Turning to temperature, [10] showed that a temperature optimal schedule could be computed in polynomial time using the Ellipsoid algorithm. Note that this is much more complicated than the simple greedy algorithm, YDS, for computing an energy optimal schedule. [10] introduces an online algorithm, BKP, that is simultaneously $O(1)$-competitive for both total energy and maximum temperature. An algorithm that is c-competitive with respect to temperature has the property that if the thermal threshold $T_{\max}$ of the device is exceeded, then it is not possible to feasibly schedule the jobs on a device with thermal threshold $T_{\max}/c$. [10] also showed that the online algorithms OA and AVR, both $O(1)$-competitive with respect to energy, are not $O(1)$-competitive for the objective of minimizing the maximum temperature. In contrast, [10] showed that the energy optimal YDS schedule is $O(1)$-competitive for maximum temperature.

Besides [10], the only other theoretical speed scaling for temperature management papers that we are aware of are [20] and [32]. In [20] it is assumed that the speed scaling policy is fixed to be: if a particular thermal threshold is exceeded then the speed of the processor is scaled down by a constant factor. Presumably chips would have such a policy implemented in hardware for reasons of self-preservation. The paper then considers the problem of how to schedule unit work tasks, that generate varying amounts of heat, so as to maximize throughput. [20] shows that the offline problem is NP-hard even if all jobs are released at time 0, and gives a 2-competitive online algorithm. [32] provides an optimal algorithm for a batched release problem similar to ours but with a different objective, minimizing the makespan, and a fundamentally different thermal model.

Surveys on speed scaling can be found in [1], [2], and [24].

4.2 PRELIMINARIES

We assume that a processor running at a speed s consumes power $P(s) = s^\alpha$, where $\alpha > 1$ is some constant. We assume that the processor can run at any non-negative real speed (using techniques in the literature, similar results could be obtained if one assumed a bounded speed processor or a finite number of speeds). The job environment consists of a collection of tasks, where each task i has an associated release time r_i, amount of work p_i, and a deadline d_i. A online scheduler does not learn about task i until time r_i, at which point it also learns the associated p_i and d_i. A schedule specifies for each time, a job to run, and a speed for the processor. The processor will complete s units of work in each time step when running at speed s. Preemption is allowed, which means that the processor is able to switch which job it is working on at any point without penalty. The deadline feasibility constraints are that all of the work on a job must be completed after its release time and before its deadline.

Our model for temperature, following [10], is Newton's law of cooling. With a power function of s^α, Newton's law of cooling gives the following differential equation to describe temperature:

$$\frac{dT(t)}{dt} = s(t)^\alpha - bT(t)$$

Where $T(t)$ is the temperature at time t, $s(t)$ is the speed at time t, and b is the processor specific cooling parameter.

Using the aforementioned models for the processor and for temperature, we will consider the online and offline problems of minimizing the maximum temperature, subject to deadline feasibility constraints.

4.3 BATCHED RELEASE

In this section, we consider the special case of the problem where all jobs are released at time 0. Instead of considering the input as consisting of individual jobs, each with a unique deadline and work, we consider the input as a series of deadlines, each with a cumulative work requirement equal to the sum of the work of all jobs due at or before that deadline.

Formally, the input consists of n deadlines, and for each deadline d_i, there is a cumulative work requirement, $w_i = \sum_{j=1}^{i} p_j$, that must be completed by time d_i. With this definition, we then consider testing the feasibility of some schedule S with constraints of the from $W(S, d_i) \geq w_i$ where $W(S, d_i)$ is the total work of S by time d_i. We call these the *work constraints*. We also have the *temperature constraint* that the temperature in S must never exceed $T_{\max}$. Without loss of generality, we assume that the scheduling policy is to always run the unfinished job with the earliest deadline. Thus, to specify a schedule, it is sufficient to specify the processor speed at each point in time. Alternatively, one can specify a schedule by specifying the cumulative work processed at each point of time (since the speed is the rate of change of cumulative work processed), or one could specify a schedule by giving the temperature at this point of time (since the speed can be determined from the temperature using Newton's law and the power function).

Before beginning with our analysis it is necessary to briefly summarize the equations describing the maximum work possible over an interval of time, subject to fixed starting and ending temperatures. First we define the function $UMaxW(0, t_1, T_0, T_1)(t)$ to be the maximum cumulative work, up to any time t, achievable for any schedule starting at time 0 with temperature exactly T_0 and ending at time t_1 with temperature exactly T_1. In [10] it is shown that:

$$UMaxW(0, t_1, T_0, T_1)(t) = \\ \left(\frac{1}{a}\right)^{\frac{1}{\alpha}} \left(\frac{T_1 - T_0 e^{-bt_1}}{e^{-bt_1} - e^{\frac{-bt_1\alpha}{\alpha-1}}}\right)^{\frac{1}{\alpha}} \left(\frac{b}{\alpha-1}\right)^{\frac{1}{\alpha}-1} \left(1 - e^{\frac{-bt}{\alpha-1}}\right) \tag{4.1}$$

The definition of the function $MaxW(0, t_1, T_0, T_1)(t)$ is identical to the definition of $UMaxW$, with the additional constraint that the temperature may never exceed $T_{\max}$. Adding this additional constraint implies that $MaxW(0, t_1, T_0, T_1)(t) \leq UMaxW(0, t_1, T_0, T_1)(t)$, with equality holding if and only if the temperature never exceeds $T_{\max}$ in the schedule for $UMaxW(0, t_1, T_0, T_1)(t)$. A schedule or curve is said to be a *UMaxW curve* if it is equal to $UMaxW(0, t_1, T_0, T_1)(t)$ for some choice of parameters. A *MaxW curve/schedule* is similarly defined. We are only concerned with $MaxW$ curves that are either $UMaxW$ curves that don't exceed $T_{\max}$ or $MaxW$ curves that end at temperature $T_{\max}$. It is shown in [10]

that these type of $MaxW$ curves have the form:

$$MaxW(0, t_1, T_0, T_{\max})(t) =$$

$$\begin{cases} UMaxW(0, \gamma, T_0, T_{\max})(t) & : t \in [0, \gamma) \\ UMaxW(0, \gamma, T_0, T_{\max})(\gamma) + (bT_{\max})^{\frac{1}{\alpha}}(t - \gamma) & : t \in (\gamma, t_1] \end{cases} \tag{4.2}$$

Here γ is the largest value of t_1 for which the curve $UMaxW(0, t_1, T_0, T_{\max})(t)$ does not exceed temperature $T_{\max}$. It is show in [10] that γ is implicitly defined by the following equation:

$$\frac{1}{\alpha - 1} T_0 e^{\frac{-b\gamma\alpha}{\alpha - 1}} + T_{\max} - \frac{\alpha}{\alpha - 1} T_{\max} e^{\frac{-b\gamma}{\alpha - 1}} = 0 \tag{4.3}$$

4.3.1 Known Maximum Temperature

In this subsection we assume the thermal threshold of the device $T_{\max}$ is known to the algorithm, and consider batched jobs. If there is a feasible schedule, our algorithm iteratively constructs schedules S_i satisfying the following invariant:

Definition 1. ***Max-Work Invariant****: S_i completes the maximum work possible subject to:*

- *For all times $t \in [0, d_n]$, the temperature of S_i does not exceed $T_{\max}$*
- *$W(S_i, d_j) \geq w_j$ for all $1 \leq j \leq i$*

By definition, the schedule S_0 is defined by $MaxW(0, d_n, 0, T_{\max})(t)$. The intermediate schedules S_i may be infeasible because they may miss deadlines after d_i, but S_n is a feasible schedule and for any feasible input an S_i exists for all i. The only reason why the schedule S_{i-1} cannot be used for S_i is that S_{i-1} may violate the i^{th} work constraint, that is $W(S_{i-1}, d_i) < w_i$. Consider the constraints such that for any $j < i$, $W(S_{i-1}, d_j) = w_j$. We call these *tight* constraints in S_{i-1}. Now consider the set of possible schedules $S_{i,j}$, such that j is a tight constraint in S_{i-1}, where intuitively during the time period $[d_j, d_i]$, $S_{i,j}$ speeds up to finish enough work so that the i^{th} work constraint is satisfied and the temperature at time d_i is minimized. Defining the temperature of any schedule S_{i-1} at deadline d_j as T_j^{i-1}, we formally define $S_{i,j}$:

Definition 2. *For tight constraint $j < i$ in S_{i-1},*

$$S_{i,j} = \begin{cases} S_{i-1} & : t \in [0, d_j) \\ UMaxW(0, d_i - d_j, T_j^{i-1}, T_i^{i,j})(t) & : t \in (d_j, d_i) \\ MaxW(0, (d_n - d_i), T_i^{i,j}, T_{\max})(t) & : t \in (d_j, d_n] \end{cases}$$

where $T_i^{i,j}$ is the solution of

$$UMaxW(0, d_i - d_j, T_j^{i-1}, T_i^{i,j})(d_i - d_j) = (w_i - w_j)$$

We show that if S_i exists, then it is one of the $S_{i,j}$ schedules. In particular, S_i will be equal to the first schedule $S_{i,j}$ (ordered by increasing j) that satisfies the first i work constraints and the temperature constraint.

Algorithm Description: At a high level the algorithm is two nested loops, where the outer loop iterates over i, and preserves the max-work invariant. If the i^{th} work constraint is not violated in S_{i-1}, then S_i is set to S_{i-1}. Otherwise, for all tight constraints j in S_{i-1}, S_i is set to the first $S_{i,j}$ that satisfies the first i work constraints and the temperature constraint. If such a $S_{i,j}$ doesn't exist, then the instance is declared to be infeasible. The following lemma establishes the correctness of this algorithm.

Lemma 14. *Assume a feasible schedule exists for the instance in question. If S_{i-1} is infeasible for constraint i, then S_i is equal to $S_{i,j}$, where j is minimized subject to the constraint that $S_{i,j}$ satisfies the first i work constraints and the temperature constraint.*

Proof. Without loss of generality, we assume $T_{\max}$ is the minimum possible maximum temperature for any schedule satisfying the first i work constraints. It is thus sufficient to show that S_i implies the existence of a feasible $S_{i,j}$, that is, an $S_{i,j}$ feasible for the first i work constraints and the temperature constraint. We show the existence of such an $S_{i,j}$ by contradiction, with two cases, either there is no deadline feasible $S_{i,j}$ or there is no $T_{\max}$ feasible $S_{i,j}$.

If there is no deadline feasible $S_{i,j}$, pick the $S_{i,j}$ with the largest j but make no assumption of temperature feasibility. We claim that this $S_{i,j}$ must be deadline feasible. Note that $S_{i,j}$ is identical to S_{i-1} on the interval $[0, d_j)$ and that on $(d_i, d_n]$ no constraints need to be satisfied,

thus $S_{i,j}$ must violate a constraint on (d_j, d_i). However both S_{i-1} and $S_{i,j}$ are $UMaxW$ curves on (d_j, d_i). $S_{i,j}$ is by construction, and S_{i-1} is because if not, it reaches a constant speed before d_i and thus reaches $T_{\max}$ by d_i. This implies $T_{\max}$ is not sufficient to satisfy constraint i, a contradiction to S_i.

As both $S_{i,j}$ and S_{i-1} are $UMaxW$ curves with the same starting temperature (T_j^{i-1}) on this interval of length $d = d_i - d_j$, it is sufficient to show that $S_{i,j}$ has completed more work than S_{i-1} at all times on the interval. In other words that,

$$UMaxW(0, d, T_j^{i-1}, T_i^{i,j})(t) > UMaxW(0, d, T_j^{i-1}, T_i^{i-1})(t) \tag{4.4}$$

It must be the case that $T_i^{i,j} > T_i^{i-1}$ as $S_{i,j}$ satisfies constraint i but S_{i-1} does not. However, if we consider equation (4.1), we can see that $UMaxW$ is a strictly increasing function of the final temperature for all times t. Thus inequality (4.4) must hold implying that $S_{i,j}$ is deadline feasible.

Because there is always a deadline feasible $S_{i,j}$, if there is no $T_{\max}$ feasible $S_{i,j}$, we consider the smallest j such that $S_{i,j}$ is deadline feasible. Because S_i must finish exactly work w_i by d_i, it must be that S_i has a lower temperature than $S_{i,j}$ at time d_i.

Consider the cumulative work accomplished by S_i and $S_{i,j}$ at all times prior to d_i and pick the latest time, call it d_v, at which the two schedules had the exact same cumulative work. We have two cases, either d_v occurs before or after d_j. Note that if there is no constraint with deadline d_v we can simply add a constraint requiring that by time d_v any feasible schedule must perform work equivalent to the cumulative work of S_i and both schedules remain unchanged.

If $d_v < d_j$, then either S_i follows a single $UMaxW$ on (d_v, d_i) or some other work curve. If S_i does follow a $UMaxW$ curve then S_i must violate a constraint because d_j is the earliest constraint such that a single $UMaxW$ curve is feasible. If S_i is not a single $UMaxW$ curve, then consider that S_v, the invariant maintaining schedule for all constraints up to and including constraint v, must be infeasible for some constraint, call it k, between d_v and d_i, and thus S_i is tight on k. However because $S_{i,j}$ satisfies all constraints up to and including i and d_v is the latest point in time at which $S_{i,j}$ and S_i differ, if S_i is tight on k, either $S_{i,j}$ is tight on k, violating the definition of d_v or $S_{i,j}$ misses a deadline, also a contradiction.

If $d_v > d_j$, then $T_j^i \geq T_j^{i,j}$ but $S_{i,j}$ has the smallest possible temperature increase on (d_j, d_i) by definition, thus S_i can't have a lower temperature by d_i and so S_i must violate $T_{\max}$, a contradiction. □

4.3.2 Unknown Maximum Temperature

In this section we again consider batched jobs, and consider the objective of minimizing the maximum temperature ever reached in a feasible schedule. Let Opt be the optimal schedule, and $T_{\max}$ be the optimum objective value. We know from the previous section that the optimum schedule can be described by the concatenation of $UMaxW$ curves $C_1, \ldots, C_{k-1}$, possibly with a single $MaxW$ curve, C_k, concatenated after C_{k-1}. Each C_i begins at the time of the $(i-1)st$ tight work constraint and end at the time of the i^{th} tight work constraint. Our algorithm will iteratively compute C_i. That is, on the i^{th} iteration, C_i will be computed from the input instance and $C_1, \ldots, C_{i-1}$. In fact, it is sufficient to describe how to compute C_1, as the remaining C_i can be computed recursively. Alternatively, it is sufficient to show how to compute the first tight work constraint in Opt.

To compute C_1, we need to classify work constraints. We say that the i^{th} work constraint is a *UMaxW constraint* if the single cumulative work curve that exactly satisfies the constraint with the smallest maximum temperature possible corresponds to equation (4.1). Alternatively, we say that the i^{th} work constraint is a *MaxW constraint* if the single cumulative work curve that exactly satisfies the constraint with the smallest maximum temperature possible corresponds to equation (4.2). We know from the last section that every work constraint must either be a $MaxW$ constraint or a $UMaxW$ constraint. In Lemma 15 we show that it can be determined in $O(1)$ time whether a particular work constraint is a $UMaxU$ constraint or a $MaxW$ constraint. In Lemma 16 we show how to narrow the candidates for $UMaxW$ constraints that give rise to C_1 down to one. The remaining constraint is referred to as the *UMaxW-winner*. In Lemma 18 we show how to determine if the $UMaxW$-winner candidate is a better option for C_1 than any of the $MaxW$ candidates. If this is not the case, we show in Lemma 19 how to compute the best $MaxW$ candidate.

Lemma 15. *Given a work constraint $W(S, d_i) \geq w_i$, it can be determined in $O(1)$ time*

whether it is a UMaxW constraint or a MaxW constraint.

Proof. For initial temperature T_0, we solve $UMaxW(0, d_i, T_0, T_i)(d_i) = w_i$ for T_i as in the known $T_{\max}$ case. Now we consider equation (4.3) for γ with $T_{\max} = T_i$:

$$\frac{1}{\alpha - 1} T_0 e^{\frac{-b\gamma\alpha}{\alpha - 1}} + T_i - \frac{\alpha}{\alpha - 1} T_i e^{\frac{-b\gamma}{\alpha - 1}} = 0$$

If we plug in d_i for γ and the left side is larger than 0, then $\gamma < d_i$ and the curve $UMaxW(0, d_i, T_0, T_i)(t)$ exceeds T_i during some time $t < d_i$, thus we have a $MaxW$ constraint. If the left side is smaller than 0, then $\gamma > d_i$ and the curve $UMaxW(0, d_i, T_0, T_i)(t)$ never exceeds T_i, implying that the constraint is a $UMaxW$ constraint. □

Lemma 16. *All of the UMaxW constraints, but one, can be disqualified as a candidate for C_1 in time $O(n)$.*

Proof. Consider any two $UMaxW$ constraints, i and j with $i < j$. We want to show that the two work curves exactly satisfying constraints i and j must be non-intersecting, except at time 0, and that we can determine which work curve is larger in constant time. This together with Lemma 15 would imply we can get rid of all $UMaxW$ constraints but one in time $O(n)$ for n constraints. For initial temperature T_0, can we can fully specify the two curves by solving $UMaxW(0, d_i, T_0, T_i)(d_i) = w_i$ and $UMaxW(0, d_j, T_0, T_j)(d_j) = w_j$ for T_i and T_j respectively. We can then compare them at all times prior to d_i using equation (4.1), i.e., $UMaxW(0, d_i, T_0, T_i)(t)$ and $UMaxW(0, d_j, T_0, T_j)(t)$.

Note that for any two $UMaxW$ curves defined by equation (4.1), a comparison results in the time dependent terms (t-dependent) canceling and thus one curve is greater than the other at all points in time up to d_i. Regardless of whether the larger work curve corresponds to constraint i or j, clearly the smaller work curve cannot correspond to the first tight constraint as the larger work curve implies a more efficient way to satisfy both constraints. To actually determine which curve is greater, we can simply plug in the values for the equations and check the values of the non-time dependent terms. The larger term must correspond to the dominating work curve. □

In order to compare the $UMaxW$-winner's curve to the $MaxW$ curves, we may need to extend the $UMaxW$-winner's curve into what we call a $UMaxW$-*extended* curve. A $UMaxW$-extended curve is a $MaxW$ curve, describable by equation (4.2), that runs identical to the $UMaxW$ constraint's curve on the $UMaxW$ interval, and is defined on the interval $[0, d_n]$. We now show how to find this $MaxW$ curve for any $UMaxW$ constraint.

Lemma 17. *Any UMaxW constraint's UMaxW-Extended curve can be described by equation (4.2) and can be computed in $O(1)$ time.*

Proof. For any $UMaxW$ curve satisfying a $UMaxW$ constraint, the corresponding speed function is defined for all times $t \geq 0$ as follows:

$$S(t) = \frac{b}{(\alpha-1)}^{\frac{1}{\alpha}} \left(\frac{T_i - T_0 e^{-bd_i}}{e^{-bd_i} - e^{\frac{-bd_i\alpha}{\alpha-1}}} \right)^{\frac{1}{\alpha}} e^{\frac{-bt}{\alpha-1}}$$

Thus we can continue running according to this speed curve after d_i. As the speed is a constantly decreasing function of time, eventually the temperature will stop increasing at some specific point in time. This is essentially the definition of γ and for any fixed γ there exists a $T_{\max}$ satisfying it which can be found by solving for $T_{\max}$ in the γ equation. To actually find the time when the temperature stops increasing, we can binary search over the possible values of γ, namely the interval $(d_i, \frac{\alpha-1}{b} \ln \frac{\alpha}{\alpha-1}]$. For each time we can directly solve for the maximum temperature using the γ equation and thus the entire $UMaxW$ curve is defined. We then check the total work accomplished at d_i. If the total work is less than w_i, then γ is too small, if larger, then γ is too large. Our binary search is over a constant-sized interval and each curve construction and work comparison takes constant time, thus the entire process takes $O(1)$ time. Once we have γ and the maximum temperature, call it T_γ, we can define the entire extended curve as $UMaxW(0, \gamma, T_0, T_\gamma)(t)$ for $0 \leq t < \gamma$ and $(bT_\gamma)^{1/\alpha} t$ for $t \geq \gamma$, in other words, $MaxW(0, \infty, T_0, T_\gamma)(t)$ with $T_{\max} = T_\gamma$. □

Lemma 18. *Any MaxW constraint satisfied by a UMaxW-Extended curve can't correspond to C_1. If any MaxW constraint is not satisfied by a UMaxW-Extended curve then the UMaxW constraint can't correspond to C_1.*

Proof. To satisfy the winning $UMaxW$ constraint exactly, we run according to the $UMaxW$-extended curve corresponding to the $UMaxW$ constraint's exact work curve. Thus if a $MaxW$ constraint is satisfied by the entire extended curve, then to satisfy the $UMaxW$ constraint and satisfy the $MaxW$ constraint it is most temperature efficient to first exactly satisfy the $UMaxW$ constraint then the $MaxW$ constraint (if it is not already satisfied). On the other hand, if some $MaxW$ constraint is not satisfied then it is more efficient to exactly satisfy that constraint, necessarily satisfying the $UMaxW$ constraint as well. □

Lemma 19. *If all UMaxW constraints have been ruled out for C_1, then C_1, and the entire schedule, can be determined in time $O(n)$.*

Proof. To find the first tight constraint, we can simply create the $MaxW$ curves exactly satisfying each constraint. For each constraint, we can essentially use the the same method as in Lemma 17 for extending the $UMaxW$ winner to create the $MaxW$ curve. The difference here is that we must also add the work of the constant speed portion to the work of the $UMaxW$ portion to check the total work at the constraint's deadline. However this does not increase the construction time, hence each curve still takes $O(1)$ time per constraint.

Once we have constructed the curves, we can then compare any two at the deadline of the earlier constraint. The last remaining work curve identifies the first tight constraint and because we have the $MaxW$ curve that exactly satisfies it, we have specified the entire optimal scheduling, including the minimum $T_{\max}$ possible for any feasible schedule. As we can have at most n $MaxW$ constraints and construction and comparison take constant time, our total time is $O(n)$. □

Theorem 2. *The optimal schedule can be constructed in time $O(n^2)$ when $T_{\max}$ is not known.*

Proof. The theorem follows from using Lemma 16 which allows us to produce a valid $MaxW$ curve by Lemma 17. We then apply Lemma 18 by comparing the $UMaxW$-winner's work at each $MaxW$ constraint. If all $MaxW$ constraints are disqualified, we've found the first tight constraint, else we apply Lemma 19 to specify the entire schedule. In either case, we've defined the schedule up to at least one constraint in $O(n)$ time. □

4.4 ONLINE ALGORITHM

Our goal in this section is to describe an online algorithm A, and analyze its competitiveness.

Algorithm Description: A runs at a constant speed of $(\ell b T_{\max})^{1/\alpha}$ until it determines that some job will miss its deadline, where $\ell = (2 - (\alpha - 1)\ln(\alpha/(\alpha - 1)))^{\alpha} \leq 2$. At this point A immediately switches to running according to the online algorithm OA. When enough work is finished such that running at constant speed $(\ell b T_{\max})^{1/\alpha}$ will not cause any job to miss its deadline, A switches back to running at the constant speed.

Before beginning, we briefly note some characteristics of the energy optimal algorithm, YDS, as well as some characteristics of the online algorithm OA. We require one main property from YDS, a slight variation on Claim 2.3 in [10]:

Claim 2. *For any speed s, consider any interval, $[t_1, t_2]$ of maximal time such that YDS runs at speed strictly greater than s. YDS schedules within $[t_1, t_2]$, exactly those jobs that are released no earlier than t_1 and due no later than t_2.*

We also need that YDS is energy optimal within these maximal intervals. This is a direct consequence of the total energy optimality of YDS. Lastly note that YDS schedules jobs according to EDF. For more on YDS, see [33] and [10]. The remaining details of YDS are not necessary for our analysis, we refer the reader to [33] and [10] for more in depth details.

For the online algorithm OA, we need only that it always runs, at any time t, at the minimum feasible constant speed for the amount of unfinished work at time t and that it has a competitive ratio of α^{α} for total energy [10].

We will first bound the maximum amount of work that the optimal temperature algorithm can perform during intervals longer than the inverse of the cooling parameter b. This is the basis for showing that the constant speed of A is sufficient for all but intervals of smaller than $1/b$.

Lemma 20. *For any interval of length $t > 1/b$, the optimal temperature algorithm completes strictly less than $(\ell b T_{\max})^{1/\alpha} \cdot (t)$ work.*

Proof. First, note that the value of γ from equation (4.3) is at most $1/b$, thus the maximum work possible, in an interval of length $t_1 > 1/b$, while not violating $T_{\max}$, is given by equation (4.2) when $T_0 = 0$. Thus at time $t = 1/b$, the optimal will be running at speed $(bT_{\max})^{1/\alpha}$ which is strictly slower than $(\ell bT_{\max})^{1/\alpha}$ therefore to prove our lemma, it is sufficient to show that $(\ell bT_{\max})^{1/\alpha} \cdot (1/b) \geq MaxW(0, 1/b, 0, T_{\max})(1/b)$. When $T_0 = 0$, we can write our desired inequality using equation (4.2) as,

$$(\ell bT_{\max})^{\frac{1}{\alpha}} \cdot (1/b) \geq UMaxW(0, \gamma, 0, T_{\max})(\gamma) + \left(\frac{bT_{\max}}{a}\right)^{\frac{1}{\alpha}} \left(\frac{1}{b} - \gamma\right) \tag{4.5}$$

First, consider just the $UMaxW$ portion. By equation (4.3), when $T_0 = 0$, the value of $\gamma = ((\alpha-1)/b) \ln(\alpha/(\alpha-1))$, and by the fact that $e^{-b\gamma} = ((\alpha-1)/\alpha)^{\alpha-1}$ in this case, using the definition of a $UMaxW$ curve from equation (4.1), we get

$$\begin{aligned}
UMaxW(0, \gamma, 0, T_{\max})(\gamma) &= \left(\frac{T_{\max}}{\frac{\alpha-1}{\alpha}^{\alpha-1} - \frac{\alpha-1}{\alpha}^{\alpha}}\right)^{\frac{1}{\alpha}} \left(\frac{\alpha-1}{b}\right)^{\frac{\alpha-1}{\alpha}} \left(1 - \frac{\alpha-1}{\alpha}\right) \\
&= (T_{\max})^{\frac{1}{\alpha}} \left(\left(\frac{\alpha-1}{\alpha}\right)^{\alpha-1} \left(1 - \frac{\alpha-1}{\alpha}\right)\right)^{-\frac{1}{\alpha}} \left(\frac{\alpha-1}{b}\right)^{\frac{\alpha-1}{\alpha}} \left(\frac{1}{\alpha}\right) \\
&= \left(\frac{1}{b}\right)^{\frac{\alpha-1}{\alpha}} (T_{\max})^{\frac{1}{\alpha}} \left(\frac{\alpha-1}{\alpha}\right)^{-\left(\frac{\alpha-1}{\alpha}\right)} (\alpha-1)^{\frac{\alpha-1}{\alpha}} \left(\frac{1}{\alpha}\right)^{\frac{\alpha-1}{\alpha}} \\
&= \left(\frac{1}{b}\right)^{\frac{\alpha-1}{\alpha}} (T_{\max})^{\frac{1}{\alpha}}
\end{aligned} \tag{4.6}$$

Now we can plug equation (4.6) back into equation (4.5) to get,

$$\begin{aligned}
(\ell bT_{\max})^{\frac{1}{\alpha}} \cdot (1/b) &\geq \left(\frac{1}{b}\right)^{\frac{\alpha-1}{\alpha}} (T_{\max})^{\frac{1}{\alpha}} + \left(\frac{bT_{\max}}{a}\right)^{\frac{1}{\alpha}} \left(\frac{1}{b} - \gamma\right) \\
\ell^{\frac{1}{\alpha}} \cdot (1/b) &\geq \frac{1}{b} + \left(\frac{1}{b} + \gamma\right) \\
\ell &\geq (2 - b\gamma)^{\alpha}
\end{aligned} \tag{4.7}$$

However, because $\gamma = \frac{\alpha-1}{b} \ln\left(\frac{\alpha}{\alpha-1}\right)$, the b constants cancel and we are left with $\ell \geq \ell$. Show that the b constants cancel out.

□

We now know that if all jobs have a lifetime of at least $1/b$, A will always run at a constant speed and be feasible, thus we have essentially handled the competitiveness of A in non-emergency periods. Now we need to consider A's competitiveness during the emergency periods, i.e., when running at speed $(\ell b T_{\max})^{1/\alpha}$ would cause A to miss a deadline. To do this, we will show that these emergency periods are contained within periods of time where YDS runs faster than A's constant speed and that during these larger periods we can directly compare A to YDS via OA. We start by bounding the maximal length of time in which YDS can run faster than A's constant speed.

Lemma 21. *Any maximal time period where YDS runs at a speed strictly greater than $(\ell b T_{\max})^{1/\alpha}$ has length $< 1/b$.*

Proof. By definition, in any of these maximal periods of length t, the YDS schedule will complete strictly more than $(\ell b T_{\max})^{1/\alpha} \cdot t$ work. By Claim 2, YDS only works on jobs both released and due in this period thus any feasible algorithm must complete strictly more than $(\ell b T_{\max})^{1/\alpha} \cdot t$ work in this period in order to be feasible. However, by Lemma 20 we know that if the optimal temperature algorithm is to complete strictly more than $(\ell b T_{\max})^{1/\alpha} \cdot t$ work then $t < 1/b$. This implies that any maximal interval during which the YDS schedule runs at a speed strictly greater than $(\ell b T_{\max})^{1/\alpha}$ is shorter than $1/b$ in length. □

We call these maximal periods in YDS *fast periods* as they are characterized by the fact that YDS is running strictly faster than $(\ell b T_{\max})^{1/\alpha}$. Now we show that A will never be behind YDS on any individual job outside of fast periods. This then allows us to describe A during fast periods.

Lemma 22. *At the beginning and ending of every fast period, A has completed as much work as the YDS schedule on each individual job.*

Proof. First note that A is feasible and that both YDS and OA use the same job selection policy. Because A runs at speed at least $(\ell b T_{\max})^{1/\alpha}$, it runs at least as fast as YDS between fast periods. Thus the lemma holds at the beginning of any fast period if it held at the end of the last fast period. By Claim 2, during any maximal fast period, YDS only works on jobs released and due during the fast period thus YDS can't complete more work than A on

any job due after the fast period. Thus the lemma must hold at the end of a fast period if it held at the beginning. From time 0 until the first fast period, A runs at least as fast as YDS, thus A has completed as much work as YDS on each individual job by the start of the first fast period. Thus the lemma holds at all times outside of fast periods. □

Lemma 23. *A switches to OA only during fast periods.*

Proof. By Lemma 22, we know that A cannot be behind YDS on work outside of a fast period. As YDS is feasible, A cannot have feasibility problems outside of these periods as it is always running at least as fast as YDS. The lemma follows from the fact that A only switches to OA when there are feasibility problems. □

We are now ready to upper bound the energy usage of A, first in a fast period, and then in an interval of length $1/b$. We then use this energy bound to upper bound the temperature of A. We use a variation on Theorem 2.2 in [10] to relate energy to temperature. We denote the maximum energy used by an algorithm, ALG, in any interval of length $1/b$, on input I, as $C[ALG(I)]$ or simply $C[ALG]$ when I is implicit. Note that this is a different interval size than used in [10]. We denote the maximum temperature of ALG on input I, similarly as $T[ALG(I)]$ or $T[ALG]$.

Lemma 24. *For any schedule S, and for any cooling parameter $b \geq 0$,*

$$\frac{aC[S]}{e} \leq T[S] \leq \frac{e}{e-1} aC[S]$$

Proof. The proof follows exactly from [10] except $c = 1/b$ instead of $\ln(2)/b$. □

Lemma 25. *A is α^α-competitive for energy in any single maximal fast period.*

Proof. By Claim 2, a fast period starts with a release and YDS will start working on the released job immediately. Because both YDS and A use EDF and because A is not behind YDS on any job by Lemma 22, A will immediately start working on the same job as YDS. Because YDS works at a speed strictly greater than $(\ell b T_{\max})^{1/\alpha}$ in the fast period, A will never be ahead of YDS for jobs in the fast period and thus A will work on the exact same jobs as YDS in the fast period. Thus we can consider the fast period as a separate input

instance and compare the energy usage of OA, YDS, and A in the fast period. A is identical to OA with a lower bound on speed and because the starting speed of YDS, the energy optimal, is above that of A which is above that of OA, A must have no worse energy usage than OA over the entire period by the convexity of the problem.

On any instance OA uses at worst α^α times the energy of YDS [10], and because A is at least as good as OA, A uses at most α^α times the minimum energy possible in any fast period. □

Lemma 26. *A uses at most $(\ell + 3e\alpha^\alpha)T_{\max}$ energy in an interval of size $1/b$.*

Proof. We can upper bound the energy used by A in a $1/b$ interval by charging it the energy used by running at $(\ell b T_{\max})^{1/\alpha}$ for the entire $1/b$ interval, plus the energy used whenever A switches to OA. From Lemma 23 we know that A only switches to OA during fast periods and by Lemma 25 A uses at most α^α times the minimum energy possible in these fast periods. As all fast periods strictly contained in a $1/b$ interval contain jobs that must be started and completed within an interval of length $1/b$, $C[OPT]$ must be at least as large as the minimum energy consumption possible for all of these periods together. Thus A uses as most $\alpha^\alpha C[OPT]$ for these fast periods. By Lemma 21, all fast periods are smaller than $1/b$ in length, thus at most two fast periods can intersect any $1/b$ interval but not be contained in it. As each of the periods are shorter than $1/b$, $C[OPT]$ must be at least as large as the minimum energy consumption of either of these fast periods, thus A uses no more than $2\alpha^\alpha C[OPT]$ energy for both of these fast periods. Thus the total energy usage of A for fast periods in a $1/b$ interval is no more than $3\alpha^\alpha C[OPT]$.

Running at speed $(\ell b T_{\max})^{1/\alpha}$ for $1/b$ uses $\ell T_{\max}$ units of energy thus the total energy usage of A in an interval of $1/b$ is no more than $(\ell T_{\max} + 3\alpha^\alpha C[OPT])$. By Lemma 24 we have that $3\alpha^\alpha C[OPT] \leq 3e\alpha^\alpha T_{\max}$, which gives a total energy usage of $(\ell + 3e\alpha^\alpha)T_{\max}$. □

Theorem 3. *A is $(\frac{e}{e-1}(\ell + 3e\alpha^\alpha))$-competitive for temperature.*

Proof. Using the fact that $T[S] \leq \frac{e}{e-1} aC[S]$ for any schedule from Lemma 24, we get that $T[A] \leq \frac{e}{e-1}(\ell + 3e\alpha^\alpha)T_{\max}$. □

4.5 IMPROVED TEMPERATURE ANALYSIS OF YDS

Using the technique from the previous section, we can improve the temperature analysis of the energy optimal offline algorithm, YDS.

Theorem 4. *YDS is* $\frac{e}{e-1}(\ell + 3e)$*-competitive for temperature, where* $15.5 < \frac{e}{e-1}(\ell + 3e) < 16.1$.

Proof. By definition of the fast periods, we know that YDS is running no faster than A outside of the fast periods. The energy usage of YDS in the fast periods follows from the exact same reasoning as that used for A, but here of course, YDS is energy optimal for those periods. □

5.0 CONCLUSION

With the introduction of speed scaling technology, operating system's scheduler now has an additional decision to make, namely, at what speed to run the processor. The fact that, in general, speed scaling technology has the property that running at a higher speed is less energy efficient, means that, a priori, it is not clear how the scheduler should decide at what speed to run the processor. If the speed is very high, a large amount of energy will be used, but if the speed is very low, then the quality of service will be very low. Thus, determining how to balance these opposing objectives of low energy usage and high quality of service has led to the re-examining of many classic scheduling problems in the context of speed scaling. The goal in these investigations is twofold. First, there is a desire to use the mechanism of speed scaling to help decrease the total energy consumption of a processor. The second goal is to build our understanding of energy usage in computer systems in order to help us reason abstractly about energy and temperature. By taking a mathematical approach, we can detach the speed scaling mechanism from the device and thus develop algorithms that apply to a range of devices, for instance, both processors and wireless network links. Reasoning about speed scaling as a mechanism, rather than a device, helps us to understand speed scaling devices in the same manner in which we currently understand space and time in abstract computing devices. With the goal of furthering these two objectives, this work has consider various classical scheduling problems in the speed scaling setting.

In chapter 2 we observed that all algorithmic speed scaling research to date has suggested that for any classical scheduling objective, $\mathcal{Q}$, with a competitive algorithm in the fixed speed setting, there is a competitive algorithm in the speed scaling setting for the objective of $\mathcal{Q}$ plus total energy. Because it seems hard to imagine that such a statement could ever be formally proven, we sought to gain confidence in the conjecture by considering one of the

most obvious classical scheduling objectives that had yet to be considered in the speed scaling setting, integer stretch. Thus we gave an algorithm, A, that was competitive for the objective of total integer stretch plus energy under a processor model which was allowed to have, essentially, an arbitrary set of possible speeds and an arbitrary set of possible corresponding powers.

In chapter 3 we considered the character of the optimal schedule, $\mathcal{S}$ for the objective $\mathcal{Q} + \beta\mathcal{E}$ for scheduling objective $\mathcal{Q}$ and total energy $\mathcal{E}$. We viewed $\mathcal{S}$ as the optimal trade-off schedule in the sense that, given the energy usage of $\mathcal{S}$, no schedule can have better quality of service $\mathcal{Q}$, and that trading an additional unit of energy (increasing $\mathcal{E}$ by 1) cannot improve the quality of service of $\mathcal{S}$ by more than β (decrease $\mathcal{Q}$ by more than β). For the particular quality of service objective of fractional flow, we sought to answer some of the natural questions one might have about these optimal trade-off schedules. Our efforts allowed us to, first, define a small set of conditions that were both necessary and sufficient for the optimality of $\mathcal{S}$:

- There is a linearly decreasing hypopower function associated with each job that maps a time to a hypopower, where the slope of the hypopower function is proportional to the density of the job, and inversely proportional to β.
- At all times, the job that is run is the job whose hypopower function is largest at that time.
- If a job is run, it is run at the hypopower specified by the hypopower function.

We then used these necessary and sufficient conditions to show that the optimal energy trade-off schedule had the following properties:

- Any schedule may be checked for these conditions, and thus for optimality, in $O(n^2)$ time.
- The optimal schedule is unique and to specify an optimal schedule, it is sufficient to specify the value of each hypopower function at the release time of the corresponding job.
- If we consider the optimal schedule as a function of β, a job's hypopower at release changes continuously as a function of β.

Using these properties we were able to develop an efficient homotopic algorithm, to compute the optimal schedule, for a modest number of jobs, by taking advantage of the facts that the optimal schedule changed continuously in β and that computing the optimal schedule for β very close to 0 was very straightforward.

We concluded chapter 3 by considering the trade-off objective of minimizing $\mathcal{Q}^{\sigma} \times \mathcal{E}$, the product of quality of service and energy, which is commonly used outside of the theoretical speed scaling literature. When $\mathcal{Q}$ was fractional flow, we showed the following:

- Perhaps counter-intuitively, we show that if the static power is zero, then the optimal solution is to either always go as fast as possible or to always go as slow as possible.
- We show that locally optimal product trade-off schedules have a nice structure similar to globally optimal sum trade-off schedules.
- We show that, for a fixed instance, the set of schedules that are optimal for the product objective (for any σ) are (a generally strict) subset of the schedules that are optimal for the sum objective (for any β).
- We show that the optimal product trade-off schedule may be a discontinuous function of σ. Thus there is unlikely to be a homotopic algorithm to compute optimal product trade-off schedules.

Based on these properties, we concluded that for the purposes of reasoning theoretically about optimal energy trade-off schedules, the sum trade-off objective is probably preferable to the product trade-off objective, as the sum trade-off objective has many more mathematically desirable properties.

Finally, in chapter 4, rather than combining energy and quality of service into a single objective, we considered the other method to handle the conflicting objectives of energy and quality of service, fixing one objective and optimizing the other. Specifically, we considered the classical deadline scheduling problem in the speed scaling setting for the objective of maximum temperature. Using Newton's Law of Cooling to model temperature, to consider a special case of the offline problem, the batch release case, i.e. when all jobs are released at the same time. Starting with the feasibility problem (i.e. we were given the max temperature, T_{max}) for the batch case, we gave a simple $O(n^2)$ algorithm. We then used the insight gained

from developing the feasibility algorithm to then develop an algorithm, for the batch case, when T_{max} is not known. We concluded chapter 4 by considering the general online problem. For the general online problem, we gave a simple competitive algorithm, A, which ran at a constant speed most of the time. The analysis of A also allowed us to improve the analysis of the energy optimal algorithm, YDS, for maximum temperature.

5.1 OPEN QUESTIONS AND FUTURE WORK

The work herein presented leaves open a number of questions and suggestions for future work. We now highlight some of these areas.

In chapter 2, we strengthened our confidence in the conjecture that states that the set of classic scheduling objectives with competitive algorithms is the same as the set of classic scheduling objectives with competitive algorithms, in the speed scaling setting, for the objective plus the total energy, under arbitrary power functions. The obvious next question is what other classic scheduling objectives, with known competitive algorithms, also have competitive algorithms in the speed scaling setting?

Chapter 3 presented a plethora of open questions, which we now enumerate.

Question 1: *Given the configuration of the optimal energy trade-off schedule (when the quality of service measure is weighted fractional flow), can the optimal schedule be computed in polynomial time?* In this context, a configuration is the sequence of jobs run on the processor as well as whether a job completes before, at, or after the release of the next job to be run (which is not necessarily the next job that was released). This seems to be the natural definition of a configuration, as the configuration and β uniquely defines a set of equations for which the initial hypopowers of the optimal schedule for that configuration are the sole (real) solution. [31] was able to use binary search to solve these equations. We were unable to compute the optimal schedule even knowing the optimal configuration. The main difficulty is that even when the power function is s^α the configuration equations yield a series of α-degree polynomial equations (in the variables hypopower, or speed, or interval length). We know of no technique to acquire a general algebraic solution to these

equations. Another natural alternative approach is to create a mathematical program from the equations, and solve them using some standard optimization algorithm. However, any formulation we examined had equations relating to work that were not convex (again, in variables hypopower, or speed, or interval length).

Question 2: *Is there an efficient algorithm to compute the optimal energy trade-off schedule (when the quality of service measure is weighted fractional flow) for $\beta + \epsilon$ given the optimal trade-off schedule for β?* As we know how to detect when the optimal configuration changes and how to find the new optimal configuration, this is closely related to the previous open question.

Question 3: *As a function of the number of jobs n, how many times can the optimal configuration change as β changes?* [31] shows that for unit jobs and when the quality of service measure is total delay, the number of configuration *changes* is $O(n^2)$. We would be quite surprised if the number of configurations changes in our setting was not also polynomially bounded, although we do not know how to prove any upper bound that is a function of n. The problem is because the initial hypopowers are not monotone in β, we do not even know how to show that a particular configuration will be optimal for a contiguous collection of β's.

Question 4: *Is there a polynomial time algorithm to verify the optimality of schedule for the objective of total (integer) delay plus energy?* Using insights from [31], one can design a polynomial time algorithm that, given a schedule S and a configuration C, can verify the optimality of S among schedules in configuration C. But we do not know how to verify that a schedule is in the optimal configuration.

Question 5: *For what quality of service measures can homotopic optimization be used to compute optimal energy trade-off schedules?* It is not completely implausible that one can characterize the natural quality of service measures where this is possible.

Finally, in chapter 4, although we made progress toward reasoning about temperature directly in the speed scaling setting, how to most effectively reason about temperature remains an open question. Reasoning directly about the optimal is difficult because the speed at which the optimal runs (as well as the optimal's temperature as a function of time) is a

complicated function, for even a single job, which has no closed form expression.

BIBLIOGRAPHY

[1] Susanne Albers. Algorithms for energy saving. In Susanne Albers, Helmut Alt, and Stefan Nher, editors, *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 173–186. Springer Berlin / Heidelberg, 2009.

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

[3] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms*, 3(4):49:1–49:17, 2007.

[4] Lachlan L.H. Andrew, Minghong Lin, and Adam Wierman. Optimality, fairness, and robustness in speed scaling designs. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 37–48, 2010.

[5] Lachlan L.H. Andrew, Adam Wierman, and Ao Tang. Optimal speed scaling under arbitrary power functions. *ACM SIGMETRICS Performance Evaluation Review*, pages 39–41, October 2009.

[6] Nikhil Bansal, David P. Bunde, Ho-Leung Chan, and Kirk Pruhs. Average rate speed scaling. In *LATIN'08: Proceedings of the 8th Latin American conference on Theoretical informatics*, pages 240–251, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[8] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 693–701, 2009.

[9] Nikhil Bansal, Ho-Leung Chan, Kirk Pruhs, and Dmitriy Katz. Improved bounds for speed scaling in devices obeying the cube-root rule. In *ICALP '09: Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, pages 144–155, Berlin, Heidelberg, 2009. Springer-Verlag.

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

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

[12] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17(6):525–532, November 1973.

[13] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[14] Ho-Leung Chan, Jeff Edmonds, Tak-Wah Lam, Lap-Kei Lee, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Nonclairvoyant speed scaling for flow and energy. *Algorithmica*, 61(3):507–517, 2011.

[15] Ho-Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. *Theory of Computing Systems*, 49(4):817–833, 2011.

[16] Ho-Leung Chan, Tak-Wah Lam, and Rongbin Li. Tradeoff between energy and throughput for online deadline scheduling. In *Workshop on Approximation and Online Algorithms*, pages 59–70. Springer Berlin Heidelberg, 2010.

[17] Sze-Hang Chan, Tak-Wah Lam, and Lap-Kei Lee. Non-clairvoyant speed scaling for weighted flow time. In *European Symposium on Algorithms*, pages 23–35. Springer Berlin Heidelberg, 2010.

[18] Sze-Hang Chan, Tak-Wah Lam, Lap-Kei Lee, Hing-Fung Ting, and Peng Zhang. Non-clairvoyant scheduling for weighted flow time and energy on speed bounded processors. In *Computing: the Australasian Theory Symposium*, pages 3–10, 2010.

[19] Ranveer Chandra, Ratul Mahajan, Thomas Moscibroda, Ramya Raghavendra, and Paramvir Bahl. A case for adapting channel width in wireless networks. *SIGCOMM Comput. Commun. Rev.*, 38(4):135–146, August 2008.

[20] Marek Chrobak, Christoph Dürr, Mathilde Hurand, and Julien Robert. Algorithms for temperature-aware task scheduling in microprocessor systems. *Sustainable Computing: Informatics and Systems*, 1(3):241 – 247, 2011.

[21] Anupam Gupta, Ravishankar Krishnaswamy, and Kirk Pruhs. Nonclairvoyantly scheduling power-heterogeneous processors. In *International Conference on Green Computing*, pages 165–173, 2010.

[22] Anupam Gupta, Ravishankar Krishnaswamy, and Kirk Pruhs. Scalably scheduling power-heterogeneous processors. In *International Colloquium on Automata, Languages and Programming, Part I*, pages 312–323, 2010.

[23] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8 –11, oct 1994.

[24] Sandy Irani and Kirk R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36(2):63–76, 2005.

[25] Tak-Wah Lam, Lap-Kei Lee, Hing-Fung Ting, Isaac K. To, and Prudence W. Wong. Sleep with guilt and work faster to minimize flow plus energy. In *International Colloquium on Automata, Languages and Programming, Part I*, pages 665–676, 2009.

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

[27] Tak-Wah Lam, Lap-Kei Lee, Isaac K. K. To, and Prudence W. H. Wong. Nonmigratory multiprocessor scheduling for response time and energy. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1527–1539, 2008.

[28] Minming Li, Andrew C. Yao, and Frances F. Yao. Discrete and continuous min-energy schedules for variable voltage processors. *Proceedings of the National Academy of Sciences of the United States of America*, 103(11):3983–3987, 2006.

[29] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[30] S. Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes E. Gehrke. Online scheduling to minimize average stretch. *SIAM Journal on Computing*, 34(2):433–452, 2005.

[31] Kirk Pruhs, Patchrawat Uthaisombut, and Gerhard J. Woeginger. Getting the best response for your erg. *ACM Transactions on Algorithms*, 4(3):38:1–38:17, June 2008.

[32] Ravishankar Rao and Sarma Vrudhula. Performance optimal processor throttling under thermal constraints. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 257–266, New York, NY, USA, 2007. ACM.

[33] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, page 374, Washington, DC, USA, 1995. IEEE Computer Society.