SCHEDULING AND ADMISSION CONTROL

BY

DAVID PATTISON BUNDE

B.S., Harvey Mudd College, 1998
M.S., University of Illinois at Urbana-Champaign, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

We present algorithms and hardness results for three resource allocation problems. The first is an abstract admission control problem where the system receives a series of requests and wants to satisfy as many as possible, but has bounded resources. This occurs, for example, when allocating network bandwidth to incoming calls so the calls receive guaranteed quality of service. Algorithms can have performance guarantees for this problem either with respect to acceptances or with respect to rejections. These types of guarantees are incomparable and algorithms having different types of guarantee can have nearly opposite behavior. We give two procedures for combining one algorithm of each type into a single algorithm having both types of guarantee simultaneously. Specifically, if we combine an algorithm that is c_A-competitive for acceptances with an algorithm that is c_R-competitive for rejections, the combined algorithm is $O(c_A)$-competitive for acceptance and $O(c_A c_R)$-competitive for rejections. If both the input algorithms are deterministic, then so is the combined algorithm. In addition, one of the combining procedures does not need to know the value of c_A and neither needs to know the value of c_R.

The second problem we consider is scheduling with rejections, a combination of scheduling and admission control. For this problem, each job comes with a rejection cost in addition to the standard scheduling parameters. The system produces a schedule for a subset of the jobs and the schedule quality is its total flow time added to the cost for all rejected jobs. We show that, even if all jobs have the same rejection cost, no online algorithm has a competitive ratio better than $(2+\sqrt{2})/2 \approx 1.707$ in general or $e/(e-1) \approx 1.582$ if all jobs have the same processing time. We also give an optimal offline algorithm for unit-length jobs with arbitrary rejection costs. This leads to a pair of 2-competitive online algorithms for unit-length jobs, one when all rejection costs are equal and one when they are arbitrary. Finally, we show that the offline problem is NP-hard even when each job's rejection cost is proportional to its processing time.

Our third problem is power-aware scheduling, where the processor can run at different speeds, with its energy consumption depending on the speeds selected. This results in a bicriteria problem with conflicting goals of minimizing energy consumption and giving a high-quality schedule. If schedule quality is measured with makespan, we give linear-time algorithms to compute all non-dominated solutions for the general uniprocessor problem and for the multiprocessor problem when every job requires the same amount of work. We also show that the multiprocessor problem becomes NP-hard when jobs can require different amounts of work. If schedule quality is measured with total flow time, we show that the optimal schedule corresponding to a particular energy budget cannot be exactly computed on a machine supporting arithmetic and the extraction of roots. This hardness result holds even when scheduling equal-work jobs on a uniprocessor. We do, however, give an arbitrarily-good approximation for scheduling equal-work jobs on a multiprocessor.

To my friends and family, without whose support this dissertation could not have been written.

Acknowledgments

I would like to thank the many people who helped and supported me during my time at UIUC. Thanks to my advisor Jeff Erickson for demonstrating success as both a researcher and teacher. I appreciate all the help and support he was willing to give to a student outside his main areas of interest and expertise. He gave me considerable guidance and our discussions directly led to some of the results contained in this dissertation. His example and comments were invaluable in teaching me how to present and write about my work.

I also owe a particular gratitude to the researchers of Sandia National Labs, where I spent several summers. My technical mentors Cindy Phillips and Vitus Leung have been excellent teachers and friends. They introduced me to scheduling and provided a great deal of help writing my first theoretical papers. In addition, they have both been valuable sources of professional advice even after I was no longer working at Sandia.

Thanks to Michael Bender of SUNY Stony Brook for all of his advice on writing and presenting research, for showing me his approach to research, and for modeling the process of building on small results. I greatly appreciate his patience and willingness to share his experiences as a young faculty member.

I thank Doug West of the UIUC Mathematics department for organizing summer research experiences for graduate students. Since much of my research was done on my own, these were a great opportunity for me to work with other students. His example also taught me about how to ask research questions and write clearly.

Many other faculty at UIUC contributed to my success. Thanks to Sariel Har-Peled for always being available and for all of his useful advice on research and my career. Thanks to Lenny Pitt and Margaret Fleck for being excellent colleagues when we co-taught CS 273 and for their advice on my job search. Thanks to Michael Heath and the numerical analysis students for welcoming

me to their seminar, exposing me to new ideas and giving me a lot of practice with presentations. Thanks to Laxmikant Kale and Marc Snir for pushing me to work on high-impact problems.

My fellow graduate students were also an important part of my experience at UIUC. I benefited from interactions with many students, but I would like to particularly thank Dan Cranston, Erin Chambers, Kevin Milans, Shripad Thite, and John Fischer for the helpful feedback they gave on my work. All of you have been great colleagues.

Finally, I thank the rest of my coauthors for sharing the research process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation considers the problem of how to share a limited resource. One paradigm for this problem is *scheduling*, in which different users are assigned different times to use the resource. In computer science, scheduling problems are typically presented in terms of processor scheduling, where a list of jobs must be executed on one or more machines. Other types of scheduling problems can be translated into this framework, with the resources becoming machines and the requests from different users becoming jobs. The specific problem depends on the requirements of each job as well as the metric used to measure schedule quality. There are many different types of job requirements and metrics, so there are far more scheduling problems than we address here; the reader is refered to surveys by Lawler et al. [60], Pruhs et al. [71], Sgall [79], and Pinedo [68].

Another paradigm for resource sharing is *admission control*. As in scheduling, the input to an admission control problem is a resource to be scheduled and a list of requests for access to that resource. Unlike in the scheduling problem, however, the requests are time-dependent so it may not be possible to satisfy all of them. The admission control algorithm must therefore choose a subset of the requests to satisfy and reject the rest. For particular applications, the algorithm may also schedule the chosen subset of requests, but the focus of admission control problems is on selecting a subset of requests to satisfy. Again, admission control is too large an area to be covered in a single dissertation. Interested readers are refered to surveys by Cottet et al. [34] and Plotkin [69].

1.1 Problems considered

Since both scheduling and admission control are huge research areas, we begin by describing the three problems addressed in this dissertation.

1.1.1 Combining admission control algorithms

Our first problem involves admission control. A typical application occurs in computer networking. Networks normally allow any machine on the network to send packets to any destination at any time. The network cannot necessarily handle all the traffic this generates, but performance typically degrades more or less evenly for all participants as packets are dropped or time out due to network congestion. This egalitarian treatment of different types of traffic causes problems for applications like streaming video, which must receive data in a timely manner to maintain picture quality. Since video frames are displayed at a fixed rate, streaming video applications essentially place a deadline on each packet of data. Packets missing their deadline might as well be dropped by the network, but not too many deadlines can be missed or the video becomes unwatchable. Requirements like the timely delivery of some percentage of an application's packets are called Quality of Service (QoS).

To guarantee QoS for networking traffic, bandwidth must be allocated to specific applications. This gives the *call control* problem, in which an allocation system receives requests to establish connections between pairs of machines in the network. Depending on the problem, each request either specifies a specific path through the network or just a pair of machines to be connected. Network bandwidth is limited so it may be necessary to reject some of these requests, but doing so allows the accepted requests to receive a bandwidth guarantee. In the discussion below, we assume that schedule quality is measured either by the number of calls accepted or the number of calls rejected, though some applications weight the calls by bandwidth or some other measure of importance.

Call control problems typically receive the connection requests over time and must decide to accept or reject each request as it arrives. Problems like this where the algorithm must make decisions based on partial information are called *online* problems, as opposed to *offline* problems, where all the input is available before any decision must be made. Many online problems cannot be solved exactly because the algorithm must commit to some decisions before knowing which choice is correct. Therefore, algorithm quality is typically measured by comparing its performance to the performance of the optimal offline algorithm.[1]

[1] Section 1.2.1 describes this technique in more detail.

OPT	Algorithm A	Algorithm R
98%	49%	96%
50%	25%	0%

Table 1.1: Incomparability of approximations with respect to acceptances and rejections. Shows percentage of requests guaranteed to be accepted by algorithms A and R depending on the percentage accepted in the optimal solution OPT. Algorithm A always accepts at least half as many requests as optimal and an algorithm R that always rejects no more than twice as many requests as optimal.

Call control algorithms can be compared to the optimal algorithm in two ways, depending on how the problem is phrased. In the *benefit problem*, the objective is to maximize the number of accepted requests, while the objective in the *cost problem* is to minimize the number of rejected requests. The optimal solution is the same for both versions of the problem, but approximations for the versions are not comparable. For example, consider an algorithm A that always accepts at least half as many requests as optimal and an algorithm R that always rejects no more than twice as many requests as optimal. If the optimal solution OPT accepts 98% of the requests, algorithm R accepts at least 96%, but algorithm A may accept only 49%. However, if OPT accepts 50% of the requests, algorithm A accepts at least 25%, but algorithm R may accept nothing. This is illustrated in Table 1.1.

In the offline setting, having a different algorithm for each measure is not a problem; given any problem instance, simulating both algorithms and taking the best solution found gives a solution good under both measures simultaneously. However, in the online setting, it is not clear how to achieve both types of guarantee simultaneously since the algorithm needs to make decisions over time. We consider this problem and give two ways to merge one algorithm of each type to get a combined algorithm having both types of guarantees.

This result is surprising since the algorithms being combined can have nearly opposite behavior. For example, consider the instance depicted in Figure 1.1, with three requests arriving for a linear network where each edge has the bandwidth to satisfy only two requests. Since each requested connection uses the edge between machines 3 and 4, at least one request must be rejected. In this situation, the algorithm of Blum et al. [22], which guarantees to reject no more than twice as many requests as optimal, rejects the two "outside" requests (those between machines 1 and 4 and

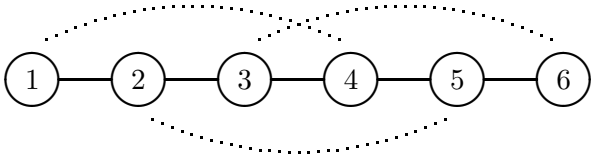

Figure 1.1: Instance where call control algorithms have opposite behavior. Dotted lines denote requested calls. Each link has bandwidth for 2 calls. The algorithm of Blum et al. [22] rejects the call between machines 1 and 4 and the call between machines 3 and 6, while the algorithm of Adler and Azar [1] rejects the call between machines 2 and 5.

machines 3 and 6). Meanwhile, the algorithm of Adler and Azar [1], which guarantees to accept at least a quarter as many requests as optimal, rejects the middle request (the one between machines 2 and 5).

1.1.2 Scheduling with rejections

Our second problem combines aspects of admission control with scheduling. In classical scheduling problems, the input is a stream of jobs, all of which must be completed. *Scheduling with rejections* adds a flavor of admission control by allowing the scheduling algorithm to reject some jobs. Unlike in pure admission control problems, however, the solution is not evaluated strictly by how many of the jobs are accepted. Rather, the accepted jobs must be scheduled and the measure of solution quality is a combination of the number of jobs accepted and the quality of the resulting schedule.

Outsourcing

One application of scheduling with rejections is planning a production schedule with outsourcing. Consider the job of a company's head of manufacturing. He receives a list of production requirements, based on either actual orders or a sales forecast. His responsibility is seeing that the required production is completed. The traditional way to do this is to schedule all the production on the company's own manufacturing equipment. Modern companies have another option, however, because some production can be outsourced to companies that operate manufacturing equipment on a contract basis. (Extreme examples of this are fabless semiconductor companies, which design chips but outsource all of their production and do not own manufacturing facilities.)

Since the company to which the work is outsourced also makes a profit, outsourcing production is probably more expensive than completing it "in house". Nevertheless, outsourcing can be beneficial if the company's own equipment would be overwhelmed by producing everything. In this case, the improvement in schedule quality may compensate for the additional cost. Thus, the head of manufacturing's task is to balance the criteria of schedule quality against the additional cost of outsourcing production.

Managing "To Do" list

Another application of scheduling with rejections is delegation. Throughout the day, you receive tasks from coworkers. Each comes with two (potentially unknown) parameters. The first is how long you will need to spend on the task to complete it. The second parameter is how long the task will take someone else, including both the time they will spend working on the task and how long before they can begin. This second parameter is the job's rejection cost, the penalty that must be paid if you delegate the task. The time until each task is completed is the amount of time before you either complete it or delegate it, plus the rejection cost for jobs that you delegated. If the objective is to minimize the total time your coworkers wait for their tasks, you again need to balance your own responsiveness with the number of tasks delegated.

1.1.3 Power-aware scheduling

Our third problem is motivated by reducing the energy consumption of computer systems. This is obviously important for battery-powered mobile devices such as laptops, where designers would like to build small devices with long battery lives. Battery life is already a problem and is likely to become more so since processor power consumption is growing at a faster rate than battery capacity [59].

Even systems that do not rely on batteries can benefit from reducing energy consumption. Computers have been estimated to use between 2% and 13% of total US electricity production [54, 50]. This is a wide range, but even the value at the low end represents a lot of money spent on electricity. Another reason to reduce computer energy consumption is that nearly all of this energy is released as heat. Modern processors are becoming harder to cool; adding a Watt of power

Speed (MHz)	Power consumed (W)
2000	89
1800	66
800	35

Table 1.2: AMD Athlon 64 power consumption at different speeds [2].

consumption to modern Intel chips adds $1–2 in packaging costs per chip [81]. The heat is also a potential health hazard since the heat generated by a processor is sufficient to fry an egg [82]. Heat generation is particularly problematic when large numbers of processors are in close proximity, such as in a supercomputer or a server farm. In fact, one study estimated that 38% of the power consumed in a data center is used for cooling [64].

Because of the importance of reducing computer power consumption, a variety of techniques have been applied to this problem [52, 65, 23, 81]. We focus on one of the more prominent approaches, *speed scaling*, the idea of conserving energy by operating more slowly.

Dynamic voltage scaling

The processor itself is the most obvious target for speed scaling. Processor speed is determined by the system clock's frequency. When the clock frequency is lowered, the processor can run at a lower voltage, which reduces the amount of energy it consumes per cycle. Because of the relationship between clock frequency and required voltage, this energy-saving technique is often called *dynamic voltage scaling*. Many modern processors implement some version of dynamic voltage scaling. For example, the AMD Athlon 64 is designed so it can run at three different frequencies, as shown in Table 1.2.

Dynamic rate scaling

Speed scaling can also be used when scheduling transmissions of a wireless device. Wireless transmissions are susceptible to noise, causing the message recipient to receive a slightly different message than the sender sent. To combat this, the different possible messages must be represented with substantially different signals so they remain distinguishable in the presence of interference.

The way to select a signal for each possible message is called an *encoding scheme*. The *rate* of the encoding scheme is the number of bits per transmission it achieves. Encoding schemes with lower rates cause the message to be sent more slowly. The energy savings come because a low rate code makes the transmission more resistant to noise, allowing transmission power to be reduced.

1.2 Definitions

Before being able to formally describe our results, we must first define some terms and concepts. For reference, an alphabetical list of definitions (including those from other sections) is provided in Appendix A.

1.2.1 Competitive analysis and approximation algorithms

Many of our results concern online algorithms. As mentioned above, online problems cannot always be solved exactly. The standard technique to measure the quality of a non-optimal online algorithm is *competitive analysis*. This involves calculating the ratio between the algorithm's performance to the performance of the offline optimal algorithm on the input instance that results in the worst ratio. Specifically, let $ALG(\sigma)$ denote the quality of the solution generated by algorithm ALG on input instance σ. Then, depending on whether the problem is a *minimization problem*, meaning the goal is to minimize the metric, or a *maximization problem*, meaning the goal is to maximize the metric, we say an algorithm A has *competitive ratio* ρ if

$$\text{minimization problem:} \quad \rho = \sup_\sigma \frac{A(\sigma)}{OPT(\sigma)} \tag{1.1}$$

$$\text{maximization problem:} \quad \rho = \sup_\sigma \frac{OPT(\sigma)}{A(\sigma)} \tag{1.2}$$

The definition is split into cases so that a lower competitive ratio always denotes a better guarantee. An optimal algorithm has competitive ratio 1. We say that an algorithm A is *ρ-competitive* if its competitive ratio is at most ρ. To calculate the competitive ratio of a randomized algorithm, replace $A(\sigma)$ with $E[A(\sigma)]$, where the expectation is over the random bits used by the algorithm.

To show that an algorithm is ρ-competitive, typically one bounds its performance in terms of parameters of the input and then bounds the optimal algorithm's performance in terms of

those same parameters. To establish a lower bound on the competitive ratio of any deterministic algorithm, one constructs a family of problem instances such that any deterministic algorithm performs poorly on at least one of them. To establish a lower bound for randomized algorithms, a powerful tool is Yao's Lemma [90], which allows one to convert a lower bound for a deterministic algorithm on randomized input into a lower bound for a randomized algorithm.

An *approximation algorithm* is an offline algorithm whose performance is bounded as a ratio of the optimal algorithm's performance. These are often used when the problem to be solved is NP-hard [46, 36]. An algorithm's *approximation ratio* ρ is defined in the same way as a competitive ratio, using Equations 1.1 and 1.2. Similarly, we say an algorithm is a ρ-approximation if its competitive ratio is at most ρ. The best type of approximation is a *polynomial-time approximation scheme (PTAS)*, an algorithm parameterized by ϵ that is a $(1+\epsilon)$-approximation with running time polynomial in the input size and $1/\epsilon$.

1.2.2 Call control

Since call control problems are both minimization problems (minimizing the number of rejected requests) and maximization problems (maximizing the number of accepted requests), we use specific terminology to discuss their competitive ratios. We say that an algorithm is *c-reject-competitive* if it is c-competitive for rejections, meaning it rejects at most c times the number of requests rejected by the optimal algorithm OPT. Similarly, we call an algorithm *c-accept-competitive* if it is c-competitive for acceptances, meaning it accepts at least $1/c$ times as many requests as OPT.

Another term we use when discussing call control algorithms is *preemption*. This is the ability to reject a request that has previously been accepted. An algorithm that uses this ability is called *preemptive* and an algorithm that does not is called *nonpreemptive*. There are both preemptive and nonpreemptive call control algorithms that are competitive with respect to acceptances, but preemption is necessary for any non-trivial competitive ratio with respect to rejections. For example, consider call control on a network of n nodes organized along a line, where node i is connected to nodes $i-1$ and $i+1$ (if they exist) and each edge having bandwidth to satisfy a single request. Suppose the first request is for the pair 1 and n. Unless the algorithm accepts this request, the input could stop, giving the algorithm an infinite competitive ratio. If the algorithm accepts this

request, however, the input can continue with requests for pairs i and $i+1$ for $i = 1, 2, \ldots, n-1$. A nonpreemptive algorithm must reject all of these requests, but the optimal solution rejects the first request and accepts these. Thus, an nonpreemptive algorithm for this problem has competitive ratio at least $n - 1$, which can be achieved by the trivial algorithm that accepts the first request and rejects all others.

1.2.3 Scheduling

Traditional scheduling problems in computer science are phrased in terms of scheduling jobs on processors. A problem is specified by giving three sets of parameters. The first gives the number of processors and the relationship between them. A *uniprocessor* problem involves scheduling a single processor while a *multiprocessor* problem involves scheduling more than one. We use m to denote the number of processors. Throughout this dissertation, we assume *identical processors*, meaning that each job takes the same amount of time on any processor. The cases where some processors are faster than others (*uniform processors*) and where jobs take arbitrarily-different amounts of time on different processors (*unrelated processors*) have also been considered in the literature, as have the problems where jobs need to be run on several processors either in a particular order (*flow shop scheduling*) or in any order (*open shop scheduling*).

The second set of parameters specifying a scheduling problem gives the type of input. A uniprocessor scheduling problem takes as input a set of n jobs $\{J_1, J_2, \ldots, J_n\}$, each of which must exclusively occupy the processor for its processing time p_i. In this dissertation, all jobs are *serial jobs*, meaning each runs on a single processor. The other possibility is *parallel jobs*, each of which must run on several processors simultaneously.

Other information is associated with each job in different scheduling problems. We consider scheduling problems that assign a *release time* r_i to each job J_i, meaning that job J_i cannot run before time r_i. For online problems, r_i is the time when the scheduling algorithm becomes aware of job J_i. Some scheduling problems also assign a *weight* w_i to each job, indicating its relative importance, or a *deadline* d_i, the time by which job J_i should (or must) be finished. Another input considered in the literature is a partial order $\prec$ of precedence constraints, where $J_i \prec J_j$ means that job J_i must be completed before job J_j can be started.

Also included as an attribute of the jobs is whether they can be preempted.[2] By default, the jobs are *nonpreemptive*, meaning that a job must run to completion once it is started. Scheduling problems with greater flexibility involve *preemptive* jobs, which can be "paused" and resumed without penalty. Intermediate in flexibility between these is a job that allows *restarts*, meaning it can be stopped and then started from the beginning at a later time.

The last set of parameters used to specify a scheduling problem is the metric used to measure performance. Generally the problem's objective is to minimize this metric, though some exceptions exist [6]. The following are the metrics mentioned in this dissertation:

- *Makespan*, $\max_i C_i^A$, where C_i^A is the time when job J_i is finished in the schedule generated by algorithm A. The quantity C_i^A is called the *completion time* of job J_i (in the schedule generated by algorithm A). The makespan is the time when the last job is finished.

- *Total completion time*, $\sum_i C_i^A$. This metric is equivalent to average completion time.[3]

- *Total weighted completion time*, $\sum_i w_i C_i^A$. This metric is equivalent to average weighted completion time. It is the same as total completion time except that different jobs are given different relative importance.

- *Total flow time*, $\sum_i F_i^A$, where $F_i^A = C_i^A - r_i$ is the *flow* of job J_i. This metric is equivalent to average flow time.[3]

- *Total weighted flow time*, $\sum_i w_i F_i^A$. This metric is equivalent to average weighted flow time. It is the same as total flow time except different jobs are given different relative importance.

- *Lateness*, $\max_i L_i^A$, where $L_i^A = C_i^A - d_i$ is the *lateness* of job J_i, the amount after its deadline it completes.

- *Tardiness*, $\max_i T_i^A$, where $T_i^A = \max\{L_i^A, 0\}$ is the *tardiness* of job J_i, essentially the same as its lateness but restricted to be non-negative.

[2] In practice, whether jobs can be preempted generally depends on the hardware, but preemptibility is considered a property of the jobs for historic reasons.

[3] Both total completion time and total flow time have been called "response time" in the literature. We avoid this terminology to prevent confusion.

The focus of this dissertation is on makespan and total flow; the other metrics are defined because they appear in related work.

The three sets of problem parameters are typically represented compactly using the $\alpha|\beta|\gamma$ notation of Graham et al. [47], with extensions by Blażewicz et al. [21], Veltman et al. [85], and Drozdowski [37]. In this notation, α describes the processors, β describes the jobs, and γ gives the performance metric. For example, $1|r_i|\sum_i F_i$ denotes uniprocessor nonpreemptive total flow with release times, while $m|r_i, pmtn|\sum_i w_i F_i$ denotes preemptive total weighted flow with release times on m processors.

In addition to the terms defined above, some notation is useful for discussing scheduling algorithms. We use S_i^A to denote the time algorithm A starts job J_i and $p_i^A(t)$ to denote the processing time remaining for job J_i at time t when scheduled by algorithm A. We call a job *active* at some time if it has been released and not yet completed and use $\delta^A(t)$ to denote the number of active jobs at time t for algorithm A. Let P denote $(\max_i p_i)/(\min_i p_i)$, the ratio between the largest and smallest processing times. Similarly, for problems involving weighted jobs, we use W to denote $(\max_i w_i)/(\min_i w_i)$, the ratio between the largest and smallest weights. Finally, let OPT denote the optimal algorithm. Table 1.3 summarizes the symbols used in scheduling problems.

Some of our results require the assumption that all processing times are the same and each arrival time is a multiple of this value. We call this special case *unit-length jobs* and assume the processing times are 1 and job arrival times are integers. This corresponds to an application with events occurring on clock ticks and/or extremely short jobs. Focusing on unit-length jobs allows us to assume that any job that starts is completed before another job arrives.

1.2.4 Scheduling with rejections

For problems where jobs can be rejected, each job J_i comes with a *rejection cost* c_i that is incurred if it is rejected. We use C to denote $(\max_i c_i)/(\min_i c_i)$, the ratio between the largest and smallest rejection costs. We say that a job is *handled* once it is either completed or rejected and use C_i^A to denote the *handling time* of job J_i under algorithm A. The reuse of C_i^A, which denotes the completion of of job J_i in traditional scheduling problems, is deliberate since C_i^A is the time when the job leaves the system in both settings. Terms from traditional scheduling based on completion

$\delta_i^A(t)$	The number of jobs active at time t in the schedule created by algorithm A.
σ_i^A	In power-aware scheduling, the speed job J_i runs at in the schedule created by algorithm A.
γ_i	In power-aware scheduling, the work requirement of job J_i.
c	The job rejection cost when all jobs have the same rejection cost.
C	The ratio between the largest and smallest job rejection costs. $(C = (\max_i c_i)/(\min_i c_i))$
c_i	The rejection cost of job J_i. Problems involving rejection are discussed in Section 1.2.4.
C_i^A	The completion time of job J_i when scheduled by algorithm A. For problems involving rejection (see Section 1.2.4), C_i^A is the handling time, the time when the job is either completed or rejected by algorithm A.
d_i	The deadline of job J_i.
F_i^A	The flow time of job J_i when scheduled by algorithm A, the amount of time it spends in the system between arrival and completion. $(F_i^A = C_i^A - r_i)$
J_i	The i^{th} job. Jobs are numbered from 1 to n.
L_i^A	The lateness of job J_i when scheduled by algorithm A, the amount after its deadline that this job is completed, i.e. $C_i^A - d_i$. Note that negative values are allowed. (T_i^A is the non-negative version.)
m	The number of processors available.
n	The number of jobs in the input.
P	The ratio between the largest and smallest job processing time. $(P = (\max_i p_i)/(\min_i p_i))$
p_i	The processing time of job J_i, the time it must run before being completed.
$p_i^A(t)$	The remaining processing time of job J_i at time t when scheduled by algorithm A.
r_i	The release time of job J_i, the time it appears in the system.
S_i^A	The start time of job J_i in schedule A.
T_i^A	The tardiness of job J_i when scheduled by algorithm A. This is a non-negative version of lateness and is calculated as $T_i^A = \max\{L_i^A, 0\}$.
W	The ratio between the largest and smallest job weights. $(W = (max_i w_i)/(\min_i w_i))$
w_i	The weight of job J_i.

Table 1.3: Symbols used in scheduling

time, such as flow $F_i^A = C_i^A - r_i$ and makespan $\max_i C_i^A$ are then derived from handling time. The following are translations of traditional scheduling metrics into the scheduling with rejections setting:

- Makespan, $\max_i C_i^A +$ cost of rejections.
- Total handling time,[4] $\sum_i C_i^A +$ cost of rejections.
- Total weighted handling time, $\sum_i w_i C_i^A +$ cost of rejections.
- Total flow time, $\sum_i F_i^A +$ cost of rejections, where $F_i^A = C_i^A - r_i$.
- Total weighted flow time, $\sum_i w_i F_i^A +$ cost of rejections.
- Lateness, $\max_i L_i^A +$ cost of rejections, where $L_i^A = C_i^A - d_i$.
- Tardiness, $\max_i T_i^A +$ cost of rejections, where $T_i^A = \max\{L_i^A, 0\}$.

1.2.5 Power-aware scheduling

Power-aware scheduling differs from the traditional scheduling problems discussed above in that the processing time of each job is no longer part of the input since the processing times are not known until the schedule is constructed. Instead, each job J_i comes with a *work requirement* γ_i. A processor running continuously at speed σ completes σ units of work per unit of time so job J_i would have processing time γ_i/σ. In general, a processor's speed is a function of time and the amount of work it completes is the integral of this function over time.

Our results concern power-aware scheduling to minimize makespan or total flow. Either of these metrics can be improved by using more energy to speed up the last job so the goals of low energy consumption and high schedule quality are in opposition. Thus, power-aware scheduling is a bicriteria optimization problem and our goal becomes finding *non-dominated schedules*. These are schedules such that no schedule simultaneously has better quality and uses less energy. A common approach to bicriteria problems is to fix one of the parameters. In power-aware scheduling, this gives two interesting special cases. Fixing energy gives the *laptop problem*, which asks "What is the best schedule achievable using a particular energy budget?". Fixing schedule quality gives

[4]Some literature calls this total completion time as in traditional scheduling.

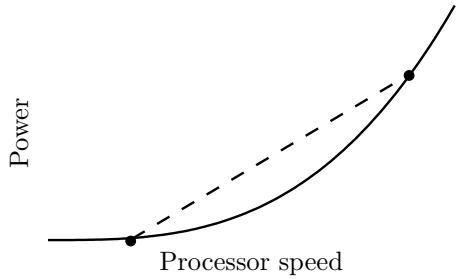

Figure 1.2: Curve power = speed^3. The dashed line demonstrates the definition of strict convexity since its interior lies above the curve.

the *server problem*, which asks "What is the least energy required to achieve a certain level of performance?". Our work in power-aware scheduling initially focused on the laptop problem, but this dissertation presents algorithms that find all non-dominated schedules.

The specification of a power-aware scheduling problem must also include the relationship between power consumption and speed. Most of our results do not assume a specific relationship between these quantities. Except where otherwise stated, we just assume that power is a continuous, strictly-convex function of processor speed. Formally, strict convexity means that the line segment between any two points on the power/speed curve lies above the curve except at its endpoints, as shown in Figure 1.2. Intuitively, strict convexity means that power increases superlinearly with speed so the "law of diminishing returns" applies to speeding up the processor to complete jobs more quickly.

Relationship between CPU power and speed

To justify these assumptions, we discuss the specific power functions corresponding to the applications described in Section 1.1.3. We begin with dynamic voltage scaling.

The specifications of processors that support dynamic voltage scaling give a list of speeds and corresponding power levels similar to Table 1.2. Since the first work on power-aware scheduling algorithms, however, researchers have assumed that the processor can run at an arbitrary speed within some range [87]. The justification for allowing a continuous range of speeds is twofold. First, choosing the speed from a continuous range is an approximation for a processor with a large number of possible speeds. Second, a continuous range of possible clock speeds is observed by

individuals who use special motherboards to overclock their computers.

Most power-aware scheduling algorithms use the model proposed by Yao et al. [91], in which the processor can run at any non-negative speed and power = speed$^\alpha$ for some constant $\alpha > 1$. (The requirement that $\alpha > 1$ makes the power function strictly convex.) This relationship between power and speed comes from an approximation of a system's switching loss, the energy consumed by logic gates switching values.

Relationship between transmission power and code rate

To demonstrate the relationship between the transmission power of a wireless device and the code rate it can use, we focus on a simple scenario where the sender can transmit at discrete times. Each signal is a real value and the noise added to this signal comes from a Gaussian distribution. The range of values that can be transmitted depends on the power so increasing the transmission power increases the information content of each signal. Specifically, if the transmission power is P and the noise has power N, the optimal information capacity of this channel is

$$C = \frac{1}{2} \lg \left(1 + \frac{P}{N} \right) \tag{1.3}$$

bits per transmission [35, pg. 242]. There are encodings to send any amount of information less than this with any desired error bound. Therefore, the energy per transmission for an optimal channel[5] is

$$P = N \left(2^{2C} - 1 \right).$$

Figure 1.3 plots this relationship, which is clearly both continuous and strictly convex. Uysal-Biyikoglu et al. [84] show that several other coding paradigms give qualitatively similar curves.

[5]The capacity given in Equation 1.3 is only achieved asymptotically so a real system would use a transmission rate $R = (1 - \epsilon)C$ for some small $\epsilon > 0$ instead of transmitting at the optimal capacity as assumed here. This does not change the shape of the curve.

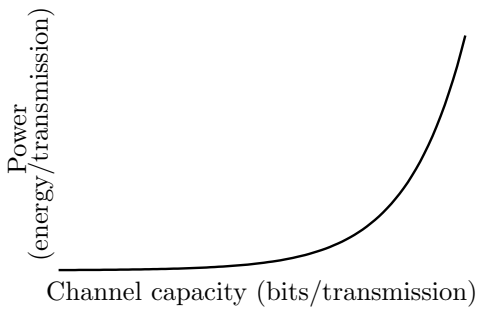

Figure 1.3: Relationship between transmission power and speed for an optimal discrete time channel with additive white Gaussian noise.

1.3 Contributions and overview of dissertation

Now we are ready to formally state our results. Our focus throughout the dissertation is on finding algorithms with provable performance bounds. To support this effort, we also give lower bounds and NP-hardness results to indicate the limits of possible performance. Our results are organized as follows:

- In Chapter 2, we talk about procedures to combine admission control algorithms. We give two procedures to combine a c_A-accept-competitive algorithm and a c_R-reject-competitive algorithm into a single preemptive algorithm that is simultaneously $O(c_A)$-accept-competitive and $O(c_A c_R)$-reject-competitive. Our procedures do not use randomness to make their decisions, so the combined algorithm is deterministic unless one of the input algorithms is randomized. Also, although the original motivation of this work and its known applications are to call control, the results apply in a very general model. The only assumption made about the resource being managed is that if it can satisfy a set S of requests, then it can also satisfy any subset of S. Thus, our procedures can be applied to other admission control problems.

  Another nice feature of our combining procedures is that they require minimal knowledge about the input algorithms, which are treated as black boxes. Neither combining procedure needs to be given the value of c_R and only one of them needs to be given the value of c_A. This ignorance of c_A and c_R gives the added benefit of making the combined algorithm's performance depend only on the behavior of algorithms A and R on the specific input sequence. For example, if algorithms A and R have log-factor competitive ratios, but happen

to be constant-competitive on the actual input sequence, then the combined algorithm is constant-competitive overall (for that input sequence).

- In Chapter 3, we discuss minimizing total flow with rejections. We give an optimal offline algorithm for the special case of unit-length jobs and show that the problem is NP-hard when jobs have varying processing times, even if the rejection cost of a job is proportional to its processing time. For the online problem with unit-length jobs, we give a 2-competitive algorithm and a family of algorithms we believe to have better performance, though we have not been able to prove a competitive ratio better than 2. We also show that, even when every job has the same rejection cost, no algorithm has a constant competitive ratio below $e/(e-1) \approx 1.582$ if all jobs have unit length or $(2+\sqrt{2})/2 \approx 1.707$ if distinct processing times are allowed.

- In Chapter 4, we discuss power-aware scheduling for makespan and total flow. We completely solve the offline uniprocessor makespan problem by giving an algorithm to find all non-dominated schedules. Its running time is linear once the jobs are sorted by arrival time. Although our results show that this is an "easy" problem in an algorithmic sense, we also show that slight variations of it are quite hard. In particular, we prove the multiprocessor version is NP-hard, even when all jobs arrive immediately. In addition, we prove that there is **NO** exact algorithm for uniprocessor total flow, even in the special case of equal-work jobs with power = speed^3. Despite these hardness results, we show how to extend uniprocessor algorithms to the multiprocessor setting when all jobs require equal work for a large class of "reasonable" metrics that includes both makespan and flow. Using this technique, we give an exact algorithm for multiprocessor makespan of equal-work jobs and an arbitrarily-good approximation for multiprocessor total flow of equal-work jobs.

Following the presentation of these results, Chapter 5 provides discussion and a summary of possible future work.

Chapter 2

Combining online algorithms

This chapter gives our two procedures to combine a c_A-accept-competitive algorithm and a c_R-reject-competitive algorithm into a single algorithm that is simultaneously $O(c_A)$-accept-competitive and $O(c_A c_R)$-reject-competitive. Section 2.1 discusses the main idea behind our procedures and also describes the previous work. Section 2.2 describes call control problems to which our combining procedure can be applied. Section 2.3 formally defines the model we use for our results. Sections 2.4 and 2.5 give our combining procedures. Finally, Section 2.6 concludes and discusses future work in combining online algorithms.

The results in this chapter were published jointly with Azar, Blum and Mansour [10]. That paper is based on conference papers by Azar et al. [11] and Bunde and Mansour [28]. In addition to the results stated here, the combined paper [10] discusses ways to combine algorithms of the same type.

2.1 Main idea and previous work

Throughout the chapter, we will use A to denote the c_a-accept-competitive algorithm and R to denote the c_R-reject-competitive algorithm. At the high level, the combining procedure uses the following simple intuitive notion: On the one hand, if the optimal solution OPT rejects only a small fraction of the requests, the reject-competitiveness of algorithm R guarantees that it accepts a large fraction of requests and thus it is accept-competitive as well. On the other hand, if OPT rejects many requests, being reject-competitive is trivial (even rejecting all requests is fine), so algorithm A is competitive by both measures. Thus, by internally simulating both algorithms, and switching between them at just the right time, we might hope to perform well in both measures. The difficulty is showing that requests rejected by one algorithm do not adversely affect the competitive ratio of

the other.

This basic idea was first used by Azar et al. [11], who gave a procedure to combine algorithms A and R into an algorithm that is simultaneously $O(c_A^2)$-accept-competitive and $O(c_A c_R)$-reject-competitive. Their procedure explicitly uses the values of both c_A and c_R, tying the combined algorithm's performance to the analysis of algorithms A and R in addition to their performance. The procedure of Azar et al. [11] also requires finding optimal solutions to the corresponding offline admission control problem at each step, a potentially-problematic requirement since many admission control problems are NP-hard. Thus, our work qualitatively improves the previous combining procedure in addition to improving its accept-competitive ratio by a factor of c_A.

2.2 Applications

Our combining procedures can be applied to several problems. One, which motivated this work, is call control on the line graph. Requests are intervals on the line and each edge in the graph has a *capacity*, which is the maximum number of accepted requests that can use that edge. Adler and Azar [1] give a constant accept-competitive algorithm for the problem and Blum et al. [22] give a constant reject-competitive algorithm for the same problem. We conclude that there is a combined algorithm that is simultaneously constant competitive for both measures. As discussed in Section 1.1.1, this is surprising since algorithms A and R can have nearly opposite behavior.

Our result can also be applied when the graph is a tree and accepted requests must be disjoint. Awerbuch et al. [8] and Awerbuch et al. [9] give $O(\log d)$-competitive randomized (nonpreemptive) algorithms for maximizing the number of accepted requests when d is the diameter of the tree. Blum et al. [22] show a constant competitive algorithm for the number of rejected requests in this case. By combining these two, we get an algorithm that is simultaneously $O(\log d)$-accept-competitive and $O(\log d)$-reject-competitive.

Another application is the admission control problem on general graphs where each edge is of logarithmic capacity and each request is for a fixed path. Awerbuch et al. [7] provide an $O(\log n)$-accept-competitive nonpreemptive algorithm and Blum et al. [22] provide an $O(\log n)$-reject-competitive (preemptive) algorithm. We conclude there are algorithms simultaneously $O(\log n)$-accept-competitive and $O(\log^2 n)$-reject-competitive.

We should remark that for many natural online problems it is impossible to achieve competitiveness in the rejection measure and hence in both measures. For example, if the online algorithm can be forced to reject a request while the offline might have not rejected any requests, then the algorithm has an unbounded competitive ratio.

2.3 Model

We assume an abstract model where one request arrives at every time unit. Either the request is served (with benefit one and cost zero), or the request is rejected (with benefit zero and cost one). A request can also be preempted, in which case its benefit is set to zero and its cost is set to one. In this abstract model, the only assumption we make about the resource constraints (which are what prevent us from accepting every request) is monotonicity: if F is a feasible set of requests, then any subset of F is feasible as well. Given a sequence σ and algorithm ALG, let $\text{BENEFIT}^{ALG}(\sigma)$ be the number of requests served by ALG and $\text{COST}^{ALG}(\sigma)$ be the number of requests rejected by ALG. By definition, the sum of benefit and cost is always the number of time steps, i.e. $\text{BENEFIT}^{ALG}(\sigma) + \text{COST}^{ALG}(\sigma) = |\sigma|$ for all algorithms ALG.

An optimal algorithm OPT can either maximize the benefit $\text{BENEFIT}^{OPT}(\sigma)$ or minimize the cost $\text{COST}^{OPT}(\sigma)$. Note that for any input sequence, the optimal schedule is identical for both maximizing benefit and minimizing cost.

We are given two algorithms. The first is a possibly randomized preemptive algorithm A that guarantees a competitive ratio of $c_A \geq 1$ for benefit. That is, for any sequence σ

$$E\left[\text{BENEFIT}^{A}(\sigma)\right] \geq \frac{1}{c_A}\text{BENEFIT}^{OPT}(\sigma) \ .$$

In addition, we are given a possibly randomized preemptive algorithm R that has a guarantee of $c_R \geq 1$ for cost. That is, for any sequence σ

$$E\left[\text{COST}^{R}(\sigma)\right] \leq c_R\text{COST}^{OPT}(\sigma) \ .$$

Notation: Given an input sequence σ, denote by σ_t the requests arriving by time t. As a

convention, the first request is number 1.

2.4 Algorithm S2

Now we describe $S2$, our first combining algorithm.[1] Internally, $S2$ simulates algorithms A and R on the input σ_t. That is, $S2$ keeps track of what requests would have been accepted or rejected had it followed algorithm A from the start, and which would have been accepted or rejected had it followed algorithm R from the start. If either algorithm is randomized, then the simulation is just of a single execution (not an average over multiple runs), and our definition of quantities such as $\text{COST}^R(\sigma)$ (e.g., Lemma 1 and Lemma 2 below) are with respect to the execution observed. At any time, $S2$ is in either an A phase or an R phase. We call the algorithm corresponding to the current phase the *phase algorithm*. Algorithm $S2$ accepts, rejects, and preempts requests in exactly the same way as the phase algorithm. Algorithm $S2$ is in an R phase if $\text{COST}^R(\sigma_t)/t \leq \tau$, where $\tau = 1/(8c_A)$, and in an A phase otherwise. Whenever $S2$ switches phases, it preempts any accepted requests that the new phase algorithm did not accept. Thus, the requests accepted by $S2$ are feasible since they are a subset of the requests accepted by the phase algorithm.

2.4.1 Analysis of rejections

We define *requests rejected because of algorithm A* to be the requests rejected or preempted during an A phase (including those rejected when switching to an A phase) and denote their number at time t with $R^A(\sigma_t)$. Thus, $\text{COST}^{S2}(\sigma_t) \leq R^A(\sigma_t) + \text{COST}^R(\sigma_t)$.

Lemma 1 *At any time t, $R^A(\sigma_t) < \text{COST}^R(\sigma_t)/\tau$.*

Proof: If $S2$ is in an A phase at time t, $\text{COST}^R(\sigma_t) > \tau t$. Since algorithm A cannot reject more than t requests, $R^A(\sigma_t) \leq t < \text{COST}^R(\sigma_t)/\tau$.

Now consider the case that $S2$ is in an R phase at time t. Let T be the last time when $S2$ was in an A phase. By the reasoning above, $R^A(\sigma_T) < \text{COST}^R(\sigma_T)/\tau$. Since $S2$ has been in an R phase since time $T+1$, $R^A(\sigma_t) = R^A(\sigma_T)$. Also, since the number of rejections of R is a non-decreasing function of time, $\text{COST}^R(\sigma_T) \leq \text{COST}^R(\sigma_t)$. Thus, $R^A(\sigma_t) < \text{COST}^R(\sigma_t)/\tau$. □

[1] We call the algorithm $S2$ because it is based on and improves over a previous algorithm "SWITCH" [11].

Since $\text{COST}^{S2}(\sigma_t) \leq R^A(\sigma_t) + \text{COST}^R(\sigma_t)$, Lemma 1 implies that

$$\text{COST}^{S2}(\sigma_t) \leq (1 + 1/\tau)\text{COST}^R(\sigma_t) \ .$$

Note that if algorithm R is randomized, then the above holds for the specific execution simulated by algorithm $S2$. Now, using the fact that algorithm R is c_R-reject-competitive, we can bound the reject-competitive ratio of $S2$ by

$$\frac{E\left[\text{COST}^{S2}(\sigma_t)\right]}{\text{COST}^{OPT}(\sigma_t)} \ \leq \ \frac{\left(1 + \frac{1}{\tau}\right) E\left[\text{COST}^R(\sigma_t)\right]}{\text{COST}^{OPT}(\sigma_t)} \ \leq \ \left(1 + \frac{1}{\tau}\right) c_R \ = \ O(c_A c_R) \ .$$

2.4.2 Analysis of acceptances

We define *requests rejected because of algorithm* R to be the requests rejected or preempted during an R phase and denote their number at time t with $R^R(\sigma_t)$.

Lemma 2 *At any time* t, $R^R(\sigma_t) \leq \text{BENEFIT}^{OPT}(\sigma_t)/(7c_A)$.

Proof: If time t is during an R phase, the lemma follows from

$$\text{BENEFIT}^{OPT}(\sigma_t) \geq \text{BENEFIT}^R(\sigma_t) \geq (1 - \tau)t \geq 7t/8$$

and $R^R(\sigma_t) \leq \text{COST}^R(\sigma_t) \leq \tau t = t/(8c_A)$.

Consider time t in an A phase. If $S2$ has not had an R phase, $R^R(\sigma_t) = 0$ so the lemma holds. Otherwise, let t' be the time at which the latest R phase ended. By the argument above,

$$R^R(\sigma_{t'}) \leq \text{BENEFIT}^{OPT}(\sigma_{t'})/(7c_A) \ .$$

Since $S2$ was in an A phase since time t',

$$R^R(\sigma_t) = R^R(\sigma_{t'}) \leq \text{BENEFIT}^{OPT}(\sigma_{t'})/(7c_A) \ .$$

Since optimal benefit is non-decreasing with the input length, $R^R(\sigma_t) \leq \text{BENEFIT}^{OPT}(\sigma_t)/(7c_A)$.
□

Now we can prove that $S2$ is $O(c_A)$-accept-competitive. We do this by bounding the number of requests accepted by both algorithms, which is a lower bound on the number of requests accepted by $S2$. Since algorithm A is c_A-accept-competitive, $E\left[\text{BENEFIT}^A(\sigma)\right] \geq \text{BENEFIT}^{OPT}(\sigma)/c_A$. By Lemma 2, algorithm R causes $R^R(\sigma_t) \leq \text{BENEFIT}^{OPT}(\sigma)/(7c_A)$ additional rejections. Thus,

$$E\left[\text{BENEFIT}^{S2}(\sigma)\right] \geq \text{BENEFIT}^{OPT}(\sigma)/c_A - \text{BENEFIT}^{OPT}(\sigma)/(7c_A) = (6/(7c_A))\text{BENEFIT}^{OPT}(\sigma)$$

and the accept-competitive ratio of $S2$ is at most $(7/6)c_A = O(c_A)$.

2.5 Algorithm RO

Now we define RO, our second combining algorithm.[2] One problem with our previous algorithm $S2$ is that, while simple, it required knowing c_A in advance. Algorithm RO is a bit more complicated but does not need to be given either of the competitive ratios as input. Internally, algorithm RO keeps times t_A and t_R, plus input prefixes σ_A and σ_R of these lengths. It maintains simulations of algorithms A and R on inputs σ_A and σ_R respectively. Whenever either of these algorithms decides to reject a request, that request is marked. Times t_A and t_R advance in phases, pausing and resuming the simulations as necessary so that $\max\{t_A, t_R\} = t$ at time t. Specifically, phase $k \geq 0$ has an R subphase, during which time t_R advances until $\text{COST}^R(\sigma_R) = 4^k$, followed by an A subphase, during which time t_A advances until $\text{BENEFIT}^A(\sigma_A) = 8 \cdot 4^k$ (note that a subphase may correspond to an empty set of requests). When a new request arrives, RO accepts it if the resulting set of accepted requests is feasible. While the resulting set is not feasible, RO preempts an arbitrary marked request (some such request must exist since $\max\{t_A, t_R\} = t$). The idea of using marks to delay rejections as long as possible is called *lazy rejection*.

2.5.1 Analysis of rejections

To analyze rejections, we first show that the algorithm maintains the invariant that $\text{BENEFIT}^A(\sigma_A) \leq 32\text{COST}^R(\sigma_R)$. (If either A or R is randomized, then this statement is with respect to the specific execution of each algorithm performed by RO.) During the first R subphase, the inequality

[2] We call this algorithm RO for Ratio Oblivious because it does not need to know the competitive ratios of the input algorithms.

holds vacuously because $\text{BENEFIT}^A(\sigma_A) = 0$. During the first A subphase, $\text{COST}^R(\sigma_R) = 1$ and $\text{BENEFIT}^A(\sigma_A) \leq 8$. Finally, during phases after the first, $\text{COST}^R(\sigma_R) \geq 4^{k-1}$ and $\text{BENEFIT}^A(\sigma_A) \leq 8 \cdot 4^k$.

Using the above inequality, we can bound $\text{COST}^A(\sigma_A)$ from above by

$$\begin{aligned} \text{COST}^A(\sigma_A) \leq t_A &= \text{COST}^{OPT}(\sigma_A) + \text{BENEFIT}^{OPT}(\sigma_A) \\ &\leq \text{COST}^{OPT}(\sigma_A) + c_A E\left[\text{BENEFIT}^A(\sigma_A)\right] \\ &\leq \text{COST}^{OPT}(\sigma_A) + 32 c_A E\left[\text{COST}^R(\sigma_R)\right] \ . \end{aligned}$$

Thus, the rejection competitive ratio of RO is at most

$$\frac{E\left[\text{COST}^A(\sigma_A) + \text{COST}^R(\sigma_R)\right]}{\text{COST}^{OPT}(\sigma_t)} \leq 1 + \frac{32 c_A E\left[\text{COST}^R(\sigma_R)\right]}{\text{COST}^{OPT}(\sigma_t)} + c_R = O(c_A c_R) \ .$$

2.5.2 Analysis of acceptances

To show that algorithm RO is $O(c_A)$-accept-competitive, we show

$$\text{BENEFIT}^{RO}(\sigma_t) \geq (1/2)\text{BENEFIT}^A(\sigma_t)$$

for all times t, all inputs, and all sequences of random bits. The desired result then follows from the competitiveness of algorithm A.

Since RO does not reject requests during the first R subphase, it is optimal and $\text{BENEFIT}^{RO}(\sigma_t) = \text{BENEFIT}^A(\sigma_t)$. The first A subphase begins when algorithm R rejects (or preempts) a request C. Algorithm RO accepts the same requests as algorithm A except possibly for C. However, $\text{BENEFIT}^{RO}(\sigma_t) \geq 1$ since RO uses lazy rejection. (This is the only part of the argument in which we use lazy rejection, but this property is necessary since otherwise RO may reject every request when it begins the first A subphase.) Thus, $\text{BENEFIT}^{RO}(\sigma_t) \geq (1/2)\text{BENEFIT}^A(\sigma_t)$. After the first subphase, $\text{COST}^R(\sigma_R) \leq (1/2)\text{BENEFIT}^A(\sigma_A)$ since $\text{COST}^R(\sigma_R) \leq 4^k$ and $\text{BENEFIT}^A(\sigma_A) \geq 8 \cdot 4^{k-1} =$

$2 \cdot 4^k$. Using this, we get

$$\begin{aligned}
\textsc{benefit}^{RO}(\sigma_t) &\geq t - \textsc{cost}^{A}(\sigma_A) - \textsc{cost}^{R}(\sigma_R) \\
&\geq (t - t_A) + \textsc{benefit}^{A}(\sigma_A) - \textsc{cost}^{R}(\sigma_R) \\
&\geq (t - t_A) + (1/2)\textsc{benefit}^{A}(\sigma_A) \ .
\end{aligned}$$

Since $\textsc{benefit}^{A}$ cannot increase faster than requests arrive, $\textsc{benefit}^{RO}(\sigma_t) \geq (1/2)\textsc{benefit}^{A}(\sigma_t)$.

2.6 Discussion

We have described procedures that take an algorithm A with competitive ratio c_A for benefit, and an algorithm R with competitive ratio c_R for cost, produce an online algorithm that simultaneously achieves competitive ratio $O(c_A)$ for benefit and $O(c_A c_R)$ for cost. We do not know if it is possible in general to do better. In particular, an ideal result in this direction would achieve $O(c_A)$ for benefit and $O(c_R)$ for cost simultaneously. There are no known lower bounds for this problem, so any lower bound better than the trivial c_A and benefit and c_R for cost would be interesting.

Chapter 3

Flow with rejections

This chapter contains our results on scheduling to minimize total flow time with rejections. Section 3.1 discusses previous work on minimizing total flow time in the traditional scheduling setting. Section 3.2 discusses the previous work on scheduling with rejections. Section 3.3 gives our lower bounds on the competitive ratio of online algorithms. Section 3.4 gives our algorithms for unit-length jobs. Section 3.5 shows that the problem is NP-complete even when each job's rejection cost is proportional to its processing time. Finally, Section 3.6 concludes with a discussion of open problems.

This chapter is based on a previously-published conference paper [26]. The results are the same as in that paper, but they are presented with more background information and improved explanations.

3.1 Previous work on minimizing total flow time

We begin our survey of previous work by discussing algorithms to minimize total flow time without rejections. This metric is related to total completion time, but by subtracting the release times for each job, it more accurately reflects the user's experience of the system; the flow time of a job is exactly the amount of time the user waits between submitting it and getting the result.

Another desirable feature of the total flow time metric is that it incrementally increases by the number of active jobs. In other words,

$$\sum_i F_i^A = \int_t \delta^A(t)\ dt,$$

where $\delta^A(t)$ is the number of active jobs at time t for algorithm A. Thus, a natural idea for

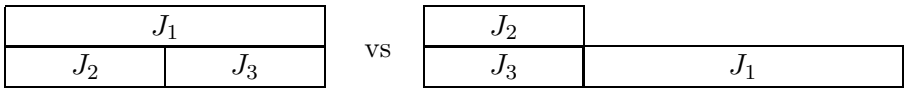

Figure 3.1: Instance where $SRPT$ falls behind in number of jobs completed

minimizing total flow time is to finish a job as soon as possible by to running the job with least processing time. When preemption is allowed, one can follow this idea precisely, an algorithm called *Shortest Remaining Processing Time* ($SRPT$). Schrage [76] and Smith [80] show that $SRPT$ has always finished at least as many jobs as any other algorithm on a uniprocessor. (Note that this claim is not true on a multiprocessor, as demonstrated in Figure 3.1.) Since the total completion time is the sum over time of the number of jobs not completed by that time, this implies that $SRPT$ is optimal for total completion time on a uniprocessor.

If preemption is not available to run $SRPT$ total flow time becomes a much harder problem, even on a single processor. Kellerer et al. [55] show that all online deterministic algorithms are $\Omega(n)$-competitive and Vestjens [86] shows that even randomized algorithms are $\Omega(\sqrt{n})$-competitive.[1] Epstein and van Stee [41] show that the latter bound holds even when the processor is allowed to restart jobs. Even offline, although Kellerer et al. [55] give an $O(\sqrt{n})$-approximation algorithm based on removing preemptions from the $SRPT$ schedule, they show that it cannot be approximated to within a factor of $n^{1/2-\epsilon}$ for any $\epsilon > 0$ unless P=NP. Bunde [24] strengthens the lower bounds to $n^{1-\epsilon}$ if the processor can only be idle when there are no active jobs. (This type of schedule is called *busy*.)

The strong inapproximability bounds in terms of n make it natural to consider approximations in terms of other parameters. One choice is P, the ratio between the largest and smallest processing times. Epstein and van Stee [42] show that *Shortest Processing Time* (SPT), the nonpreemptive version of $SRPT$ that just starts the shortest available job whenever the processor becomes free, is a $P+1$ approximation. They also give an $\Omega(\min\{n, P\})$ lower bound for deterministic algorithms.[2]

[1] The proof presented by Vestjens [86] seems to be incorrect [24]. The result is correct, however, as shown by Epstein and van Stee [41].

[2] The actual approximation result by Epstein and van Stee [42] was that an algorithm called *Revised Levels* running on lm processors could get a solution with at most $O(\min\{P^{1/l}, n^{1/l}\})$ times the total flow time of the optimal algorithm running on m processors. The lower bound was $\Omega(\min\{n^{1/l}, P^{1/l}\}/(12l)^l)$ in the same setting. The quoted results follow by plugging in $l = 1$ and noticing that *Revised Levels* on a single processor is SPT.

Bunde [25] improves these results by showing that SPT is $(P+1)/2$-competitive for nonpreemptive total flow time and that no nonpreemptive algorithm can achieve a constant ratio better than this. Even with randomization and the ability to restart jobs, Epstein and van Stee [41] show that any algorithm is $\Omega(\sqrt{P})$-competitive.[3] When a randomized algorithm is required to be busy, Bunde [24] shows that it cannot have a constant competitive ratio better than $(P+6)/8$. Even offline, Kellerer et al. [55] shows that total flow time cannot be approximated to within a factor of $P^{1/3-\epsilon}$ for any constant $\epsilon > 0$ unless P=NP.[4]

Minimizing total flow time on a multiprocessor is hard even with preemption. Leonardi and Raz [63] show that any online algorithm is $\Omega(\log(n/m))$-competitive and $\Omega(\log P)$-competitive using the idea in Figure 3.1. They also show that $SRPT$ is $O(\log(n/m))$-competitive and $O(\log P)$-competitive, meaning is it still within a constant factor of being the best algorithm. (Leonardi [62] gives a simpler version of the upper bound.) Phillips et al. [67] show that $SRPT$ becomes optimal for the multiprocessor case when given processors that are $(2 - 1/m)$ times as fast.

When preemption is not allowed, Epstein and van Stee [42] show that every deterministic multiprocessor algorithm is $\Omega(P/m)$-competitive.[5]

In the offline setting, many total flow problems are NP-hard since the equivalent total completion problem is known to be NP-hard. (Note that the optimal solution is the same for these two metrics.) In particular, both minimizing either metric is NP-hard on a multiprocessor [38], without preemptions [61], and with weighted jobs [58]. Unlike with total completion time, where Afrati et al. [3] give a PTAS for all these problems, however, total flow time is also hard to approximate. Leonardi and Raz [63] show that nonpreemptive multiprocessor total flow time with $m \leq n^{\epsilon/4}$ cannot be approximated within a factor of $n^{1/3-\epsilon}$ or $P^{1/3-\epsilon}$ for any $\epsilon > 0$ unless P=NP.[6] Bunde [24] strengthens these lower bounds to $n^{1-\epsilon}$ and $P^{1-\epsilon}$ if the schedules are required to be busy.

The best known algorithm for multiprocessor total flow time without preemptions is due to

[3]This result is not specifically claimed by Epstein and van Stee [41]; it is achieved by a slight modification of their proof that every randomized algorithm is $\Omega(\sqrt{n})$-competitive even with restarts.

[4]This result is not specifically claimed by Kellerer et al. [55]; it is achieved by noticing that $P = O(r^2 g) = O(r^3 k^2) = O(n^{3/2})$ in their $n^{1/2-\epsilon}$ lower bound argument.

[5]This result is not specifically claimed by Epstein and van Stee [42]; it is achieved by noticing that $P = 2n/m$ in their $\Omega(n/m^2)$ lower bound argument.

[6]The result in terms of P is not specifically claimed by Leonardi and Raz [63]; it is achieved by noticing that $P = O(mr^2 g) = O(mr^3 k^2) = O(r^3 k^3) = O(n)$ in their argument that no algorithm can achieve a approximation factor of $\Omega(n^{1/3-\epsilon})$.

<table>
<tr><th></th><th></th><th>preemptive</th><th>nonpreemptive</th></tr>
<tr><td rowspan="2">1</td><td>online</td><td rowspan="2">1 [Sc68,Sm78]</td><td>$\Omega(\sqrt{n})$ [86, 41]
$\Omega(\sqrt{P})$ [41]
$\leq \frac{P+1}{2}$ [25]
deterministic alg.: $\Omega(n)$ [55]
$> \frac{P+1}{2} - \epsilon$ [25]</td></tr>
<tr><td>offline</td><td>$O(\sqrt{n})$
$\omega(n^{1/2-\epsilon})$ if P $\neq$ NP [55]</td></tr>
<tr><td rowspan="2">m</td><td>online</td><td>$\Theta(\log \min\{P, n/m\})$ [63]</td><td>deterministic alg.: $\Omega(n/m^2)$ [42]</td></tr>
<tr><td>offline</td><td>$O(\log \min\{P, n/m\})$ [63]</td><td>$O(\sqrt{n/m}\log(n/m))$
$\omega(n^{1/3-\epsilon})$ if P $\neq$ NP [63]
$\omega(P^{1/3-\epsilon})$ if P $\neq$ NP</td></tr>
</table>

Table 3.1: Approximibility of total flow time

Leonardi and Raz [63]. It is an $O(\sqrt{n/m}\log(n/m))$-approximation and is based on removing the preemptions from the *SRPT* schedule using techniques similar to those applied by Kellerer et al. [55] in the uniprocessor setting.

Table 3.1 summarizes the known results on total flow time.

3.1.1 Weighted total flow time

Unlike total flow, weighted total flow time cannot be solved optimally in the online setting even with preemption. Epstein and van Stee [41] show that deterministic preemptive algorithms cannot be better than 2-competitive and that randomized preemptive algorithms cannot be better than 4/3-competitive. (The 4/3 bound was independently discovered by Chekuri et al. [31].) When preemption is not allowed, Epstein and van Stee [41] show that every algorithm is $\Omega(n)$-competitive. They also show that every algorithm is $\Omega(\sqrt{W})$-competitive when restarts are not allowed. In the multiprocessor setting, Chekuri et al. [31] show that randomized preemptive algorithms are $\Omega(\min\{\sqrt{P}, \sqrt{W}, \sqrt[4]{n/m}\})$-competitive.

As with total flow time, it is useful to think about the incremental increase in weighted flow over time when designing algorithms. An algorithm is said to be *locally c-competitive* if the weight of its active jobs is at most c times the optimal weight of active jobs at all times. Becchetti et al. [20] show that local c-competitiveness is a necessary and sufficient condition for an algorithm to be c-

	preemptive	nonpreemptive
1	$\geq 4/3$ [41, 31] $O(\log W)$ [15] $O(\log n + \log P)$ if n,P, and W known [15] $O(\log^2 P)$ if P known [31] deterministic alg: ≥ 2 [41]	$\Omega(n)$ [41] $\Omega(\sqrt{W})$ [41]
m	$\Omega\left(\min\{\sqrt{P}, \sqrt{W}, \sqrt[4]{n/m}\}\right)$ [31]	

Table 3.2: Competitiveness of online weighted total flow time

competitive. Bansal and Dhamdhere [15] use this idea to give a k-competitive algorithm, where k is the number of distinct weights. By rounding job weights to powers of two, their algorithm becomes $O(\log W)$-competitive. If the values of n, P, and W are known, Bansal and Dhamdhere [15] also give an $O(\log n + \log P)$-competitive preemptive algorithm. If only the value of P is known, Chekuri et al. [31] give an $O(\log^2 P)$-competitive preemptive algorithm.

Table 3.2 summarizes the known results on the competitiveness of online algorithms for total weighted flow.

In the offline setting, Chekuri et al. [31] give a $(2+\epsilon)$-approximation algorithm that runs in $n^{O(\log^2 n)}$ time assuming that the processing times are polynomial in n. Chekuri et al. [30] give a PTAS when $P = O(1)$ and an $(1+\epsilon)$-approximation algorithm with running time $O(n^{O(\log W \log P/\epsilon^2)})$.

3.2 Previous work on scheduling with rejections

Scheduling with rejections is a relatively new research area. The work most relevant to our results is by Bansal et al. [14], included in Bansal's dissertation [13], who pose the problem of minimizing total flow time with rejections. They show that no randomized algorithm is better than $\sqrt[4]{n}$ or $\sqrt{C}$-competitive when the rejection costs are arbitrary. (Recall that C is the ratio between the largest and smallest rejection costs.) Therefore, the bulk of the paper focuses on the case when all rejection costs are the same. For this case, they give a 2-competitive algorithm and a lower bound of 3/2 that holds even for unit-length jobs. They also mention a $(1+\epsilon)$-approximation with running time $n^{O(\log n/\epsilon^2)}$ for the offline problem.

Bansal et al. [14] also consider the weighted problem. Using ideas of Bansal and Dhandhere [15], they generalize their 2-competitive algorithm to the weighted problem by rounding weights to

powers of 2. This leads to an $O(\log^2 W)$-competitive algorithm if all jobs in a weight class have the same rejection cost. (Recall that W is the ratio between the largest and smallest weights.) They also give an algorithm for general rejection costs that is $O(\frac{1}{\epsilon}(\log W + \log C)^2)$-competitive if it runs on a processor that is $(1+\epsilon)$ times faster than the optimal solution's processor.

Engels et al. [40] consider offline algorithms to minimize a related metric, weighted handling time. They give optimal algorithms for some special cases when all jobs arrive at time 0 on a single machine and approximation algorithms for scheduling multiple unrelated machines. They also show that minimizing weighted handling time is NP-complete even when all jobs arrive at time 0, have weight equal to their processing time, and have the same rejection cost. This implies that minimizing weighted flow time is NP-complete under the same conditions.

Scheduling with rejections was introduced by Bartal et al. [19], who tried to minimize makespan on a nonpreemptive multiprocessor. Seiden [77] gives better bounds by allowing preemption and Hoogeveen [48] considers the offline problem with preemption. Sengupta [78] considers scheduling with rejections to minimize lateness and tardiness.

3.3 Lower bounds

Now we give lower bounds on the competitive ratio when all jobs have the same rejection cost, which we denote with c. In our proofs, we use OPT to refer to the optimal algorithm and ALG to refer to a hypothetical algorithm whose competitive ratio is better than our bounds.

3.3.1 When all jobs have unit length

Our first lower bound applies to unit-length jobs. It is a generalization of 3/2-competitiveness bound due to Bansal et al. [14]. The instance in their bound has two parts. The first is a stream of jobs, one released at each unit time. The second is a single "extra" job, released at time 0. The stream continues until the algorithm rejects a job, which we may assume to be the extra one. Their lower bound of 3/2 comes from the tradeoff between rejecting the extra job quickly, in which case OPT finishes all jobs quickly and there is little to amortize the rejection cost against, and rejecting the extra job later, in which case OPT rejects it immediately and the algorithm's flow time is too large.

Our lower bound instance consists of a stream of jobs arriving one per time unit and k extra jobs released at time 0. We will show that the algorithm cannot achieve a competitive ratio better than $e/(e-1)$ because it must wait until a time $\tau_{k,i}c$ to make the i^{th} rejection, but that delaying the rejections to these times yields the same competitive ratio. We define $\tau_{k,i}$ recursively as follows:

$$\begin{aligned} \tau_{k,0} &= 0 \\ \tau_{k,i} &= \frac{i + \sum_{j=1}^{i-1} \tau_{k,j} - (i-1)\tau_{k,i}}{(k+1)/(e-1)}. \end{aligned} \tag{3.1}$$

Subtracting $\tau_{k,i-1}$ from $\tau_{k,i}$ and solving for the difference gives the following alternate definition:

$$\tau_{k,i} = \sum_{j=1}^{i} \frac{1}{(k+1)/(e-1) + (j-1)} \tag{3.2}$$

In this paper, we use the following observations about $\tau_{k,k}$:

Lemma 3 *For all values of k, $\tau_{k,k} < 1$.*

Lemma 4 $\lim_{k\to\infty} \tau_{k,k} = 1$.

These lemmas have technical and unenlightening proofs, which appear in Appendix B. Instead, we proceed with the lower bound:

Theorem 5 *No deterministic algorithm for flow time with rejections is $(e/(e-1)-\epsilon)$-competitive for any constant $\epsilon > 0$ even when all jobs have unit length and the same rejection cost.*

Proof: Suppose to the contrary that ALG is $(e/(e-1)-\epsilon)$-competitive. Choose k large enough so $\tau_{k,k} \geq 1 - \epsilon/2$, which is possible by Lemma 4. We consider the instance described above consisting of a stream of jobs, one released at each unit time, and k extra jobs released at time 0. All jobs have unit length and rejection cost $c = 6k^2/\epsilon$.

Let time t_i be when ALG makes its i^{th} rejection, assuming the stream continues until time t_i. We consider two cases. The first is $t_i < \tau_{k,i}c$ for some smallest value i. In this case, the last job of the stream arrives at time t_i. ALG has cost at least $ic + \sum_{j=1}^{i} t_j + (k-i+1)t_i + \binom{k+1-i}{2} > ic + \sum_{j=1}^{i} t_j + (k-i+1)t_i$. OPT has cost at most $(k+1)t_i + \binom{k+1}{2}$. If $i = 1$, the competitive ratio

is at least $ec/((e-1)c+k^2) > e/(e-1)-\epsilon$ and we are done. If $i>1$, the cost of OPT is at most $(k+1)t_i + \binom{k+1}{2} \leq (k+1)t_i + k^2 \leq (k+1)t_i(1+\epsilon/(3(e-1))) \leq (k+1)t_i(1+\epsilon(e-1)/e)$. Algebraic manipulation of the ratio between these bounds leads to a contradiction:

$$\begin{aligned}
\frac{ic+\sum_{j=1}^{i} t_j + (k-i+1)t_i}{(k+1)t_i(1+\epsilon(e-1)/e)} &= \frac{1}{1+\epsilon(e-1)/e} + \frac{ic+\sum_{j=1}^{i-1} t_j - (i-1)t_i}{(k+1)t_i(1+\epsilon(e-1)/e)} \\
&\geq \frac{1}{1+\epsilon(e-1)/e} + \frac{ic+\sum_{j=1}^{i-1} \tau_{k,j}c - (i-1)\tau_{k,i}c}{(k+1)\tau_{k,i}c(1+\epsilon(e-1)/e)} \\
&= \frac{e}{(e-1)(1+\epsilon(e-1)/e)} \\
&> \frac{e}{e-1} - \epsilon.
\end{aligned}$$

In the second case, $t_i \geq \tau_{k,i}c$ for each i. In this case, the last job of the stream arrives at time t_k. The cost of ALG is $kc + \sum_{i=1}^{k} t_i + t_k + 1 \geq kc + \sum_{i=1}^{k-1} \tau_{k,i}c + 2t_k + 1$. OPT can immediately reject all k extra jobs, so its cost is at most $kc + t_k + 1$. Again, manipulation of the ratio between these bounds leads to a contradiction:

$$\begin{aligned}
\frac{kc+\sum_{i=1}^{k} \tau_{k,i} + t_k + 1}{kc + t_k + 1} &\geq \frac{kc+\sum_{i=1}^{k} \tau_{k,i}c + \tau_{k,k}c + 1}{kc + \tau_{k,k}c + 1} = 1 + \frac{\sum_{i=1}^{k} \tau_{k,i}}{kc + \tau_{k,k} + 1} \\
&> 1 + \frac{\sum_{i=1}^{k} \tau_{k,i}}{k+1+1/c} > 1 + \frac{\sum_{i=1}^{k} \tau_{k,i}}{(k+1)(1+\epsilon/6)} \\
&\geq 1 + \frac{1}{(e-1)(1+\epsilon/6)} - \frac{k\epsilon}{2(k+1)(1+\epsilon/6)} \\
&\geq 1 + \frac{1-\epsilon/3}{e-1} - \epsilon/2 \\
&> \frac{e}{e-1} - \epsilon.
\end{aligned}$$

□

3.3.2 When jobs have arbitrary processing times

For jobs with arbitrary processing times, we prove the following stronger lower bound:

Theorem 6 *When jobs have uniform rejection cost and arbitrary processing time, no deterministic online algorithm has a constant competitive ratio better than* $(2+\sqrt{2})/2 \approx 1.707$.

Again, the best previously-known lower bound was 3/2, due to Bansal et al. [14], which applied to unit-length jobs with uniform rejection costs.

Proof: Let $\xi = (2 + \sqrt{2})/2$. Suppose to the contrary that a deterministic algorithm ALG is $(\xi - \epsilon)$-competitive. Without loss of generality, we assume $1/\epsilon$ is an integer. Let $c = 2\xi/\epsilon$.

Using these values, we give an input instance on which ALG has competitive ratio worse than $\xi - \epsilon$. At time 0, release a job with processing time $c/\xi = 2/\epsilon$. Starting at time $c/\xi - 1$, release a job with processing time 1 every unit time until ALG rejects a job at some time t. If $t < c/\xi - 1$, the instance has only one job and the competitive ratio is at least $c/(c/\xi) = \xi > \xi - \epsilon$. Otherwise, the competitive ratio is

$$\frac{2t - (c/\xi - 1) + c + 1}{\min\{t + 1, c\} + t + 1 - (c/\xi - 1)}.$$

This is minimized when $t = c - 1$, so this ratio is at least

$$\frac{3c - c/\xi}{2c - c/\xi + 1}.$$

Since the ratio is at most 2, this is greater than

$$\frac{3c - c/\xi - 2}{2c - c/\xi} > \xi - 2/c > \xi - \epsilon.$$

□

Intuitively, the bound is improved by the flow time incurred before ALG rejects the long job.

3.4 Algorithms for unit-length jobs

Now we give algorithms for the special case of unit-length jobs. By restricting ourselves to this case, we give an optimal offline algorithm and a 2-competitive online algorithm for jobs with arbitrary rejection costs. The only previously-known algorithm for arbitrary rejection costs was the online algorithm of Bansal et al. [14] that is $O(\frac{1}{\epsilon}(\log W + \log C)^2)$-competitive if it runs on a processor that is $(1 + \epsilon)$ times faster than the optimal solution's processor.

3.4.1 Optimal offline algorithm

We begin by observing which jobs an offline optimal algorithm rejects. We call $r_i + c_i$ the *rank* of job J_i and denote it with $rank(J_i)$.

Lemma 7 *If OPT rejects some job J_i when scheduling unit-length jobs, then no job with rank lower than $rank(J_i)$ is active any time during the interval $[r_i, r_i + c_i)$.*

Proof: Suppose to the contrary that OPT rejects some job J_i but completes a job J_j during $[r_i, r_i + c_i)$, with $r_j + c_j < r_i + c_i$. Consider schedule OPT' that is identical to OPT except that OPT' rejects J_j at time r_j and runs J_i in the time freed up, i.e. starting at time $C_j^{OPT} - 1$ and ending at C_j^{OPT}. Jobs J_i and J_j contribute $c_i + C_j^{OPT} - r_j$ to OPT and $C_i^{OPT'} - r_i + c_j$ to OPT'. By construction, $C_i^{OPT'} = C_j^{OPT}$ and the cost of every job other than J_i and J_j is the same. Thus, $cost(OPT') - cost(OPT) = -r_i + c_j - (c_i - r_j) = r_j + c_j - (r_i + c_i) < 0$. Therefore, schedule OPT' is strictly better than OPT, a contradiction. □

Implicit in the proof of this lemma is that a rejected job can be swapped for a job of equal rank without affecting the total cost. Let a *normalized optimal solution* be an optimal solution that rejects the fewest number of jobs and always runs the active job with highest rank, breaking ties with the job number. Such an optimal solution always exists because the order in which unit-length jobs are run does not affect the flow.

Lemma 7 leads us to the algorithm Offline Highest Rank First (*HRF–OFF*). At all times, *HRF–OFF* runs the active job with highest rank, again using job numbers to break ties. It rejects any job whose flow time equals its rejection cost, i.e., any job not completed by time equal to its rank. Each of *HRF–OFF*'s rejections occurs at the rejected job's arrival time.

Theorem 8 *Algorithm* HRF–OFF *is optimal for unit-length jobs with arbitrary rejection costs.*

Proof: Let OPT be a normalized optimal solution. By construction, OPT and *HRF–OFF* can only differ if they reject different jobs. Consider the first time this occurs. Since both algorithms always reject jobs when they arrive, this time is the arrival time r_i of some job J_i.

Suppose *HRF–OFF* rejects job J_i and OPT does not. This is the first difference between the schedules so OPT and *HRF–OFF* have the same active jobs just prior to time r_i. Also, Lemma 7

implies that OPT does not reject jobs ranked above J_i arriving between times r_i and C_i^{OPT}. Since both OPT and *HRF–OFF* run jobs in order of decreasing rank, they have the same higher-ranked active jobs throughout this interval. *HRF–OFF* is busy with higher-ranked jobs until time $r_i + c_i$ because it rejects job J_i. Thus, the flow time of job J_i in OPT is greater than c_i, a contradiction.

Now suppose OPT rejects job J_i and *HRF–OFF* does not. Let $t = C_i^{HRF-OFF} - 1$ be the time *HRF–OFF* starts job J_i. At time t, *HRF–OFF* has completed all jobs ranked above J_i that have arrived because it runs jobs in order of decreasing rank. (None are rejected since a job is rejected exactly when it cannot be completed by time equal to its rank and $t < rank(J_i) \leq$ "higher" rank.) OPT also runs jobs in order of decreasing rank so it has also completed these jobs. By Lemma 7, OPT has no lower-ranked active jobs at time t since $r_i \leq t < rank(J_i) = r_i + c_i$. Thus, it has no active jobs at time t and running job J_i then instead of rejecting it is a valid schedule. This schedule has fewer rejections than OPT and no worse cost, a contradiction. □

3.4.2 2-competitive online algorithm

HRF–OFF makes most of its decisions online, but is an offline algorithm since running it online requires the ability to reject jobs retroactively. We call the algorithm that rejects jobs when their flow time equals their rejection cost Online Highest Rank First (*HRF–ON*) since it can run online.

By Theorem 8, *HRF–ON* is optimal if each rejection is made retroactively at the job's release time. Since each rejected job J_i accumulates flow time c_i before rejection, we get the following:

Theorem 9 HRF–ON *is 2-competitive for unit-length jobs having arbitrary rejection costs.*

In fact, *HRF–ON* is not $(2-\epsilon)$-competitive for any constant $\epsilon > 0$ even if we assume it rejects jobs once it is clear their flow time will exceed their rejection cost. Consider the following instance where all jobs have rejection cost c: $\sqrt{c}$ jobs (the *main group*) arrive at time 0 and another job arrives at each time i for $i = 0, \ldots, c-1$ (the *stream*). The algorithm keeps all the main group jobs until time $c - \sqrt{c}$, at which point it rejects them one at a time, one per unit time. Thus, these jobs incur cost $\sqrt{c}(c-\sqrt{c}) + \binom{\sqrt{c}}{2} + c\sqrt{c} = 2c^{3/2} - c/2 + \sqrt{c}/2$. Jobs in the stream run as they arrive, for combined flow time and cost c. Thus, the total cost is $2c^{3/2} + c/2 + \sqrt{c}/2$. The optimal algorithm rejects the main group immediately and runs the stream as it arrives, for total cost $c^{3/2} + c$. Thus, the competitive ratio approaches 2 as c increases.

3.4.3 Online algorithm when all jobs have rejection cost c

Let the *pseudoschedule* of a schedule be the modification created by keeping each rejected job J_i active for c_i time units. Intuitively, these jobs wait until being instantly finished or rejected for free. Note that the pseudoschedule's total flow is the cost of the schedule from which it is made. When this schedule is an optimal schedule, we call the result an *optimal pseudoschedule*. The optimal pseudoschedule can be generated by an online simulation of *HRF–OFF* since *HRF–OFF* is optimal and decides whether to reject each job J_i within time c_i of its release.

Using this simulation, we define a family of algorithms called *ratio rejecting*. A member of this family is denoted $RR(\rho)$ for a constant $\rho > 1$, though we omit ρ when discussing a general property of the family or when its value is clear from context. Let $\textsc{cost}(EOPT, t)$ and $\textsc{cost}(RR(\rho), t)$ be the costs incurred up to time t by the simulation and $RR(\rho)$, respectively. $RR(\rho)$ always runs the most recently-arrived job. It rejects the active job with earliest arrival time whenever

$$\frac{\textsc{cost}(RR(\rho), t) + c}{\textsc{cost}(EOPT, t)} \leq \rho. \tag{3.3}$$

Since $\textsc{cost}(EOPT, t)$ is a lower bound on the optimal cost, this condition ensures that the competitive ratio is at most ρ when $RR(\rho)$ makes a rejection. This observation quickly leads to the following lemma:

Lemma 10 *$RR(\rho)$ is ρ-competitive if it handles each job within time c of its release.*

Proof: Let OPT be an optimal pseudoschedule. We claim that each job is active in OPT at least as long as in RR. For jobs that OPT rejects, this follows immediately from our assumption on job handling times since these jobs are active in OPT for time c after their release. Suppose to the contrary that there exists a job that OPT runs before RR. Let J_i be the job with earliest completion time in RR having this property. Let J_j be the job RR completed immediately before J_i. Because RR runs the job with latest arrival time, $r_j \geq r_i$. OPT does not reject job J_j since it does not reject job J_i. Also, because it uses the same criteria to select which job to run, OPT must have completed job J_j before running job J_i. Thus, $C_j^{OPT} < C_i^{OPT} \leq C_j^{RR}$, contradicting our assumption that job J_i was the first job that OPT runs before RR.

Now the lemma follows by considering how the ratio $\text{COST}(RR, t)/\text{COST}(EOPT, t)$ changes over time. Since each job is active in OPT at least as long as in RR, $\text{COST}(EOPT, t)$ increases at least as fast as $\text{COST}(RR, t)$ except when RR rejects a job. Thus, the ratio declines toward 1 except when RR rejects a job. By definition, however, RR only rejects a job when doing so does not raise the ratio above ρ. Therefore, the ratio lies between 1 and ρ at all times. Since this ratio at the end of the input is exactly the competitive ratio, the competitive ratio of $RR(\rho)$ is at most ρ. □

Using this, we can show that $RR(2)$ is 2-competitive:

Theorem 11 *$RR(2)$ is 2-competitive for unit-length jobs all having rejection cost c.*

Proof: Call all jobs that $RR(\rho)$ does not run within time c of their release *extra* jobs. (Note that this includes all jobs RR rejects.) We show that all extra jobs are rejected within time c of their release; the result then follows from Lemma 10. First we consider the simple case when all extra jobs arrive at a single time. Without loss of generality, assume this time is 0. Let k be the number of extra jobs and let OPT be a normalized optimal solution.

Let t_i be the time of RR's i^{th} rejection. For notational convenience, we define $t_0 = 0$. We show that the time between t_{i-1} and t_i is at most $c/(k+i)$. The result then follows from Lemma 3 since the i^{th} rejection occurs before $\tau_{k,i}$ as defined in Equation 3.2. Consider how the ratio $\text{COST}(RR, t)/\text{COST}(EOPT, t)$ changes between times t_{i-1} and t_i. All k extra jobs contribute to the denominator since they remain active in the pseudoschedule even if OPT rejects them. Only $k-(i-1) = k+1-i$ of them contribute to the numerator since $i-1$ have already been rejected. In addition to the extra jobs, there is at least one other job active at all times or RR would work on extra jobs. This job and any other non-extra job contributes equally to the numerator and denominator since OPT and RR process jobs in the same order. Thus, the rejection occurs latest if $k+2-i$ jobs contribute to the numerator and $k+1$ contribute to the denominator. Therefore, if RR has not made the i^{th} rejection before time $t_{i-1} + c/(k+i)$, the largest possible value of the ratio in Equation 3.3 is

$$\frac{\text{COST}(RR, t_{i-1}) + (k+2-i)c/(k+i) + c}{\text{COST}(EOPT, t_{i-1}) + (k+1)c/(k+i)} \leq 2$$

and RR makes the i^{th} rejection at that time.

It remains to show that each extra job is rejected within time c of its release when extra jobs arrive at several times. Suppose to the contrary that an input instance exists in which some job J is not handled within time c of its release. Since extra jobs arriving after job J only hasten its rejection, we assume the extra jobs arriving with J are the last extra jobs and the only ones not rejected within time c of their arrival. We also assume that non-extra jobs arrive one per unit time since deviations from this also hasten rejections. Call each set of extra jobs arriving concurrently a *group*. We give a modification of the instance that reduces the number of groups in such a way that every rejection after the arrival of the second group occurs no earlier. This suffices since repeatedly applying this transformation to our initial instance creates an instance with a single group of extra jobs that are not all rejected within time c, which we have already shown cannot occur.

Suppose the first two groups of extra jobs arrive at times 0 and t, respectively. Let k be the size of the first group and let i be the number of these jobs rejected by time t. Our transformation removes all jobs arriving before time t and adds $k - i$ jobs to the second group. We show the rejections occur later in this modified instance by showing that $\textsc{cost}(RR, t)$ grows at least as quickly and $\textsc{cost}(EOPT, t)$ grows more slowly. The value of $\textsc{cost}(RR, t)$ grows at least as quickly at time t since exactly the same number of jobs are active in RR. The faster growth continues after time t because slower growth of $\textsc{cost}(EOPT, t)$ delays RR's rejections. To show that $\textsc{cost}(EOPT, t)$ grows more slowly, consider how the first group of extra jobs contribute to it after time t in the original instance. We showed above that an extra group with k jobs causes i rejections by time $\sum_{j=1}^{i} c/(k+j) \leq ci/k$ after its arrival. Thus, $t \leq ci/k$. Since the first group has k jobs contributing to $\textsc{cost}(EOPT, t)$ until time c, their total contribution after time t is at least $ck(1-i/k) = c(k-i)$. This is exactly the total contribution of the $k - i$ jobs we replaced them with. In addition, the replacement jobs contribute until time $t + c$ rather than stopping at time c. Since the contribution of the replacement jobs is no larger and it is more dispersed, $\textsc{cost}(EOPT, t)$ grows more slowly. □

3.5 NP-completeness of minimizing flow with rejections

Turning to the offline problem, we show that a restricted version of the problem is NP-complete. This is the first offline hardness result for the unweighted problem, answering an open question

posed by Bansal [13].

Theorem 12 *Minimizing total flow time with rejections is NP-complete even when the cost of rejecting a job is proportional to its processing time.*

Note that this theorem implies that minimizing the sum of completion times with rejection is also NP-complete since the same schedule is optimal for total flow time and sum of completion times. In fact, our proof uses ideas of Du et al. [38], who proved that it is NP-complete to find a multiprocessor preemptive schedule minimizing the sum of completion times. Rejected jobs in our schedule are equivalent to jobs running on a second processor in theirs.

Proof: The problem is in NP because once the jobs to reject are specified, the others are run using *SRPT* [14]. To show hardness, we give a reduction from PARTITION [46]:

> PARTITION: Given a multiset $A = \{a_1, a_2, \ldots, a_n\}$, does there exist a partition of A into A_1 and A_2 such that $\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i$?

Let $B = \sum_{i=1}^{n} a_i$. We assume B is even since otherwise such a partition cannot exist. We also assume $n \geq 5$ since small instances of PARTITION can be quickly decided. We create a scheduling instance with $n+2$ jobs where each job's rejection cost is $5/2$ times its processing time. The first n jobs are *small jobs*, each created from a member of A. These are released periodically. In addition, one large job is released immediately and another arrives later. The instance is constructed so that a solution to PARTITION corresponds to a schedule rejecting the jobs created from members of A_1 and completing all other jobs by the arrival time of the second large job. Specifically, we create job J_i created from a_i, where $r_i = 2(i-1)B$, $p_i = 2a_i$, and $c_i = 5a_i$. For the large jobs, $p_{n+1} = p_{n+2} = 2nB$, $c_{n+1} = c_{n+2} = 5nB$, $r_{n+1} = 0$, and $r_{n+2} = 2nB + B$. We ask for a schedule with cost at most $(4n + 9/2)B$. Figure 3.2 illustrates the instance.

Given a solution to PARTITION, we construct a schedule having this cost exactly as described above. We reject the jobs created from elements of A_1 and run the other small jobs as they arrive. Job J_{n+1} runs interspersed with these, completing at time $2nB + B$ so job J_{n+2} runs as soon as it arrives. This gives a total cost of exactly $(4n + 9/2)B$.

It remains to show how to construct sets A_1 and A_2 from a schedule with cost at most $(4n + 9/2)B$. Clearly, the schedule does not reject job J_{n+1} or job J_{n+2}. Let A_1 contain the elements of A

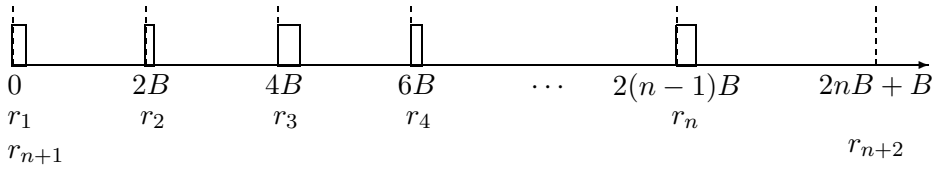

Figure 3.2: Instance showing that minimizing flow with rejections is NP-hard. Depicts the release time of every job and shows each small job running at its release time.

that created rejected jobs and A_2 contain the other members of A. We assume that each accepted small job J_i runs immediately after arriving since otherwise doing so produces a schedule with no greater cost. Similarly, we assume that job J_{n+1} completes before job J_{n+2} starts. If $\sum_{a_i \in A_2} a_i > B/2$, then job J_{n+1} completes after the arrival of J_{n+2} and so the cost is at least $5\sum_{a_i \in A_1} a_i + 2\sum_{a_i \in A_2} a_i + \left(2nB + 2\sum_{a_i \in A_2} a_i\right) + \left(2nB + 2\sum_{a_i \in A_2} a_i - B\right) = 4nB + 4B + \sum_{a_i \in A_2} a_i > (4n + 9/2)B$. (Recall that $\sum_{a_i \in A_1} a_i + \sum_{a_i \in A_2} a_i = B$.) However, if $\sum_{a_i \in A_2} a_i < B/2$, then the cost is at least $5\sum_{a_i \in A_1} a_i + 2\sum_{a_i \in A_2} a_i + \left(2nB + 2\sum_{a_i \in A_2} a_i\right) + 2nB = 4nB + 5B - \sum_{a_i \in A_2} a_i > (4n + 9/2)B$. Thus, the elements of A_1 and A_2 each sum to $B/2$. □

3.6 Discussion

Although we have made progress on several parts of this problem, there remain areas for further work. We are most interested in closing the gap between the lower bound of $e/(e-1) \approx 1.582$ and the various 2-competitive algorithms for unit-length jobs with rejection cost c. We believe that RR can be better than 2-competitive; certainly there is slack for a single group of extra jobs, which $RR(2)$ finishes by approximately $0.7c$. There is also still a gap between our lower bound of $(2 + \sqrt{2})/2 \approx 1.707$ and the 2-competitive algorithm of Bansal et al. [14] for jobs with arbitrary processing times and rejection cost c. In addition, it would be interesting to know if randomized algorithms can beat the competitive ratios given in Theorems 5 and 6. In the offline setting, we have shown that one form of the problem is NP-complete, but it is not known whether the problem remains NP-complete when all jobs have rejection cost c. Another open question is whether either the full problem or this special case admit a PTAS.

Chapter 4

Power-aware scheduling

The chapter contains our results on power-aware scheduling. Section 4.1 describes previous work in power-aware scheduling. Section 4.2 gives our uniprocessor algorithm to find all non-dominated schedules for makespan. Section 4.3 shows that the laptop problem cannot be exactly solved for total flow time. Section 4.4 extends the uniprocessor results to give our multiprocessor algorithms for equal-work jobs and shows that the general multiprocessor laptop problem is NP-hard for makespan. Finally, Section 4.5 discusses future work.

A conference paper with the results of this chapter will appear subsequent to the publication of this dissertation [27].

4.1 Previous work on power-aware scheduling

The work most closely related to ours is due to Uysal-Biyikoglu, Prabhakar, and El Gamal [84], who consider the problem of minimizing the energy of wireless transmissions. As with our work, the only assumption required by their algorithms is that the power function is continuous and strictly convex. They give a quadratic-time algorithm to solve the server version for makespan. Thus, our algorithm represents an improvement by running faster and also finding all non-dominated schedules rather than just solving the server problem.

Several variations of the wireless transmission problem have also been studied. El Gamal et al. [44] consider the possibility of packets with different power functions. They give an iterative algorithm that converges to an optimal solution. They also show how to extend their algorithm to handle the case when the buffer used to store active packets has bounded size and the case when packets have individual deadlines. Keslassy et al. [56] claim a non-iterative algorithm for packets with different power functions and individual deadlines when the inverse of the power function's

derivative can be represented in closed form. (Their paper gives the algorithm, but only a sketch of the proof of correctness.)

Another transmission scheduling problem, though one that does not correspond to a processor scheduling problem, is to schedule multiple transmitters. If we require that only one transmitter operate at a time, another extension of the iterative algorithm of Uysal-Biyikoglu and El Gamal [83] converges to the optimal solution. In general, however, there may be a better solution in which transmitters sometimes deliberately interfere with each other. Uysal-Biyikoglu and El Gamal [83] give an iterative algorithm to find this solution.

In the processor scheduling literature, the work most closely related to the results in this chapter is due to Pruhs, van Stee, and Uthaisombut [73]. They consider the laptop problem version of minimizing makespan for jobs having precedence constraints where all jobs are released immediately and power = speed$^\alpha$. Their main observation, which they call the *power equality*, is that the sum of the powers of the machines is constant over time in the optimal schedule. They use binary search to determine this value and then reduce the problem to scheduling on related fixed-speed machines. Previously-known [33, 29] approximations for the related fixed-speed machine problem then give an $O(\log^{1+2/\alpha} m)$-approximation for power-aware makespan. This technique cannot be applied in our setting because the power equality does not hold for jobs with release dates.

Minimizing the makespan of tasks with precedence constraints has also been studied in the context of project management. Speed scaling is possible when additional resources can be used to shorten some of the tasks. Pinedo [68] gives heuristics for some variations of this problem.

A special case of precedence constraints is periodic synchronization barriers. Kappiah et al. [53] describe a system to slow down lightly-loaded processors, with the goal of eliminating idle time caused by synchronization.

The only previous power-aware algorithm to minimize total flow time is by Pruhs, Uthaisombut, and Woeginger [72], who consider scheduling equal-work jobs on a uniprocessor. In this setting, they observe that jobs can be run in order of release time and then prove the following relationships between the speed of each job in the optimal solution:

Theorem 13 ([72]) *Let $J_1, J_2, \ldots, J_n$ be equal-work jobs ordered by release time. In the schedule*

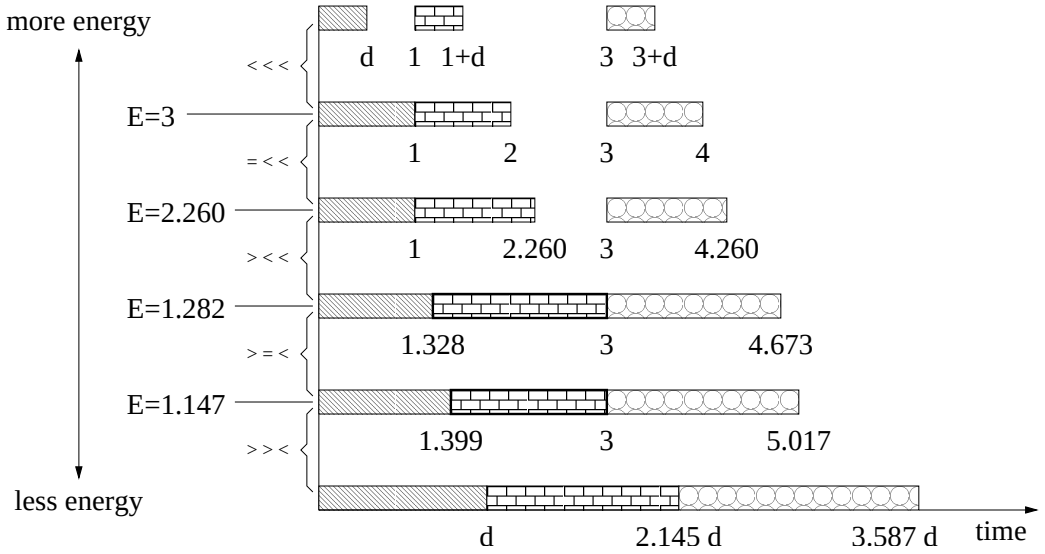

Figure 4.1: Non-dominated schedules for power-aware total flow time scheduling instance with $r_1 = 0$, $r_2 = 1$, $r_3 = 3$, $\gamma_1 = \gamma_2 = \gamma_3 = 1$, and power = speed^3, taken from Pruhs et al. [72].

OPT minimizing total flow time for a given energy budget where power = speed$^\alpha$, the speed σ_i of job J_i (for $i \neq n$) obeys the following:

- *If $C_i^{OPT} < r_{i+1}$, then $\sigma_i = \sigma_n$.*
- *If $C_i^{OPT} > r_{i+1}$, then $\sigma_i^\alpha = \sigma_{i+1}^\alpha + \sigma_n^\alpha$.*
- *If $C_i^{OPT} = r_{i+1}$, then $\sigma_n^\alpha \leq \sigma_i^\alpha \leq \sigma_{i+1}^\alpha + \sigma_n^\alpha$.*

These relationships, together with observations about when the schedule changes configuration, give an algorithm based on binary search that finds an arbitrarily-good approximation for either the laptop or the server problem. Figure 4.1, reproduced from Pruhs et al. [72], shows the optimal configurations for an instance with 3 jobs. The configurations are labeled with strings of symbols $<$, $>$, and $=$. Each symbol gives the relationship between the completion time of a job and the release of its successor. For example, a $<$ in the first position means that job J_1 completes before time r_2.

The algorithm of Pruhs et al. [72] actually gives more than the solution to a single problem. It can be used to plot the exact tradeoff between total flow time and energy consumption for optimal

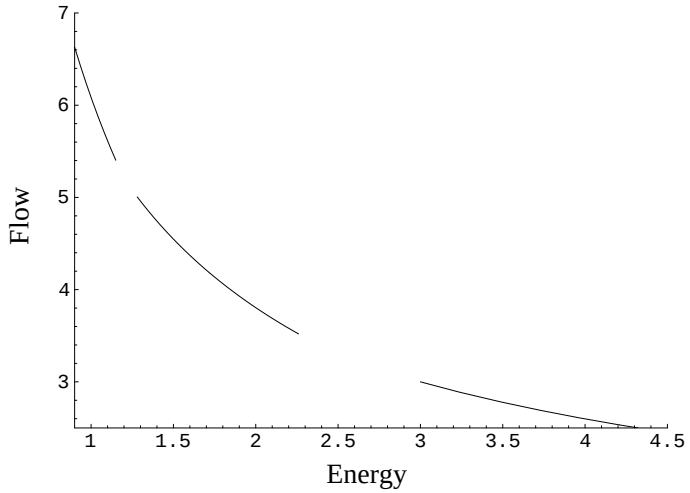

Figure 4.2: Relationship between energy and total flow time for instance with $r_1 = 0$, $r_2 = 1$, $r_3 = 3$, $\gamma_1 = \gamma_2 = \gamma_3 = 1$, and power = speed^3, taken from Pruhs et al. [72].

schedules in which the third relationship of Theorem 13 does not occur. Figure 4.2, also reproduced from Pruhs et al. [72], shows this tradeoff for configurations without one job arriving exactly as another is released, i.e. those for which the third relationship of Theorem 13 does not occur. Our impossibility result in Section 4.3 shows that the difficulty caused by the third relationship cannot be avoided.

Albers and Fujiwara [4] propose a variation of the problem with the objective of minimizing the sum of energy consumption and total flow. When power = speed$^\alpha$, they show that every online nonpreemptive algorithm is $\Omega(n^{1-1/\alpha})$-competitive using an input instance where a short job arrives once the algorithm starts a long job. Their main result is an online algorithm for the special case of equal-work jobs whose competitive ratio is at most $8.3e(1+\phi)^\alpha$, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the Golden Ratio. This competitive ratio is constant for any fixed α, but very large; for $\alpha = 3$, its value is approximately 405. They also give an arbitrarily-good approximation for the offline problem with equal-work jobs and suggest another possible online algorithm. Bansal, Pruhs, and Stein [18] analyze this suggested online algorithm using a potential function and show it is 4-competitive. They also show that a related algorithm has a competitive ratio around 20 for weighted jobs.

The idea of power-aware scheduling was proposed by Weiser et al. [87], who use trace-based simulations to estimate how much energy could be saved by slowing the processor to remove idle time. Yao et al. [91] formalize this problem by assuming each job has a deadline and seeking the minimum-energy schedule that satisfies all deadlines. They give an optimal offline algorithm and

propose two online algorithms. They show one is $(2^{\alpha-1}\alpha^\alpha)$-competitive, i.e. it uses at most $2^{\alpha-1}\alpha^\alpha$ times the optimal energy. Bansal et al. [16] analyze the other, showing it is α^α-competitive. Bansal et al. [16] also give another algorithm that is $(2(\alpha/(\alpha-1))^\alpha e^\alpha)$-competitive.

Power-aware scheduling of jobs with deadlines has also been considered with the goal of minimizing the CPU's maximum temperature. Bansal et al. [16] propose this problem and give an offline solution based on convex programming. Bansal and Pruhs [17] analyze the online algorithms discussed above in the context of minimizing maximum temperature.

A different variation is to assume that the processor can only choose between discrete speeds. Chen et al. [32] show that minimizing energy consumption in this setting while meeting all deadlines is NP-hard, but give approximations for some special cases.

Another algorithmic approach to power management is to identify times when the processor or parts of it can be partially or completely powered down. Irani and Pruhs [52] survey work along these lines as well as approaches based on speed scaling.

4.2 Makespan scheduling on a uniprocessor

Our first result is an algorithm to find all non-dominated schedules for uniprocessor power-aware makespan. We begin by solving the laptop problem for an energy budget E. Let OPT be an optimal schedule for this problem, i.e. OPT has minimum makespan among schedules using energy E.

4.2.1 Algorithm for laptop problem

To find OPT, we establish properties it must satisfy. Our first property is due to Yao, Demers, and Shenker [91], who observed that the speed does not change during a job or energy could be saved by running that job at its average speed.

Lemma 14 ([91]) *Each job runs at a single speed in OPT.*

We use σ_i^A to denote the speed of job J_i in schedule A, omitting the schedule when it is clear from context.

The second property allows us to fix the order in which jobs are run.

Lemma 15 *Without loss of generality, OPT runs jobs in order of their release times.*

To simplify notation, we assume the jobs are indexed so $r_1 \leq r_2 \leq r_3 \leq \ldots \leq r_n$.

Proof: We show that any schedule can be modified to run jobs in order of their release times without changing the energy consumption or makespan. If schedule A is not in this form, then A runs some job J_i followed immediately by some job J_j with $j < i$. We change the schedule by starting job J_j at time S_i^A, the time schedule A starts job J_i, and job J_i after job J_j completes at time $S_i^A + p_j^A$. The speed of each job is the same as in schedule A so the energy consumption is unchanged. The interval of time when jobs J_i and J_j are running is also unchanged so the transformation does not affect makespan. The resulting schedule A' is legal since each job starts no earlier than its release time. In particular, $r_j \leq r_i \leq S_i^A = S_j^{A'}$ and $r_i \leq S_i^A < S_i^{A'}$. □

The third property is that OPT is not idle between the release of the first job and the completion of the last job.

Lemma 16 *OPT is not idle between the release of job J_1 and the completion of job J_n*

Proof: Suppose to the contrary that there is an interval of time between times r_1 and C_n^{OPT} during which OPT specifies idle time. If OPT is idle before running job J_1, modify the schedule to start the first job at its release, finishing it at the same time as in OPT. Otherwise, there is a job J_i running before the idle time. In this case, slow job J_i so that it completes at the end of the idle time. In either case, our modification means some job runs more slowly. This change saves energy, which can be used to speed up job J_n and lower the makespan, contradicting the optimality of OPT. □

Stating the next property requires a definition. A *block* is a maximal substring of jobs such that each job except the last finishes after the arrival of its successor. For brevity, we denote a block with the indices of its first and last jobs. Thus, the block with jobs $J_i, J_{i+1}, \ldots, J_{j-1}, J_j$ is block (i, j). The fourth property is the analog of Lemma 14 for blocks.

Lemma 17 *In OPT, jobs in the same block run at the same speed.*

Proof: If the lemma does not hold, we can find two adjacent jobs J_i and J_{i+1} in the same block of OPT with $\sigma_i \neq \sigma_{i+1}$. Let ϵ be a positive number less than the amount of work remaining in

job J_i at time r_{i+1}. Consider changing the schedule by running ϵ work of J_i at speed σ_{i+1} and ϵ work of J_{i+1} at speed σ_i. Since the block contains the same amount of work at each speed, the makespan is unchanged and the same amount of energy is used. By construction, this change does not cause the schedule to violate any release times. Job J_i does not run at a constant speed, however, contradicting Lemma 14. □

Lemma 17 shows that speed is a property of blocks. In fact, if we know how OPT is broken into blocks, we can compute the speed of each block. The definition of a block and Lemma 16 mean that block (i, j) starts at time r_i. Similarly, block (i, j) completes at time r_{j+1} unless it is the last block. Thus, any block (i, j) other than the last runs at speed $(\sum_{k=i}^{j} \gamma_k)/(r_{j+1} - r_i)$. To compute the speed of the last block, we subtract the energy used by all the other blocks from the energy budget E. We choose the speed of the last block to exactly use the remaining energy.

Using the first four properties, an $O(n^2)$-time dynamic programming algorithm can find the best way to divide the jobs into blocks. To improve on this, we establish the following restriction on allowable block speeds:

Lemma 18 *The block speeds in OPT are non-decreasing.*

Proof: Suppose to the contrary that OPT runs a block (i, j) faster than block $(j+1, k)$. Let $\epsilon > 0$ be less than the amount of work in either block. We modify the schedule by running ϵ of the work in each block at the other block's speed. This does not change when the pair of blocks complete or how much energy they consume since the same amount of work is run at each speed. The modified schedule is valid since no job starts earlier than in OPT. Thus, we have created another optimal schedule, but it runs block (i, j) at two speeds, contradicting either Lemma 14 or Lemma 17. □

It turns out that OPT is the only schedule having all the properties given by Lemmas 14–18.

Lemma 19 *For any energy budget, there is a unique schedule having the following properties:*

1. *Each job runs at a single speed*
2. *Jobs are run in order of release time*
3. *It is not idle between the release of job J_1 and the completion of job J_n*

4. Jobs in the same block run at the same speed

5. The blocks speeds are non-decreasing

Proof: Suppose to the contrary that A and B are different schedules obeying all five properties and consuming the same amount of energy. Since each schedule is determined by its blocks, A and B must have different blocks. Without loss of generality, suppose the first difference occurs when job J_i is the last job in its block for schedule A but not for schedule B. We claim that every job indexed at least i runs slower in schedule B than in schedule A. Since energy consumption increases with speed, this implies that schedule B uses less energy than schedule A, a contradiction.

In fact, we prove the strengthened claim that every job indexed at least i runs slower and finishes later in schedule B than in schedule A. First, we show this holds for job J_i. Job J_i ends its block in schedule A but not in schedule B so $C_i^B > r_{i+1} = C_i^A$. Since each schedule begins the block containing job J_i at the same time and runs the same jobs before job J_i, job J_i runs slower in schedule B than schedule A.

Now we assume that the strengthened claim holds for jobs indexed below j and consider job J_j. Since each job $J_i, \ldots, J_{j-1}$ finishes no earlier than its successor's release time in schedule A, each finishes after its successor's release time in schedule B. Thus, none of these jobs ends a block in schedule B and schedule B places jobs J_i and J_j in the same block, which implies $\sigma_j^B = \sigma_i^B$. Speed is non-decreasing in schedule A so $\sigma_i^A \leq \sigma_j^A$. Therefore, $\sigma_j^B = \sigma_i^B < \sigma_i^A \leq \sigma_j^A$ so job J_j runs slower in schedule B than in schedule A. Job J_j also finishes later because job J_{j-1} finishing later implies that job J_j starts later. □

Because only OPT has all five properties, we can solve the laptop problem by finding a schedule with the properties. For this task, we propose an algorithm $IncMerge$. This algorithm maintains a tentative list of blocks, initially empty. Each block knows its speed, calculated as described above from the release time of the next job (including jobs not yet added to the schedule) or the energy budget. Jobs are added to the schedule one at a time in order of their release times. When a new job is added, it starts in its own block. Then, while the last block runs slower than its predecessor, the last two blocks are merged. Assuming the input jobs are already sorted by release time, $IncMerge$ runs in $O(n)$ time since each job ceases to be the first job of a block once.

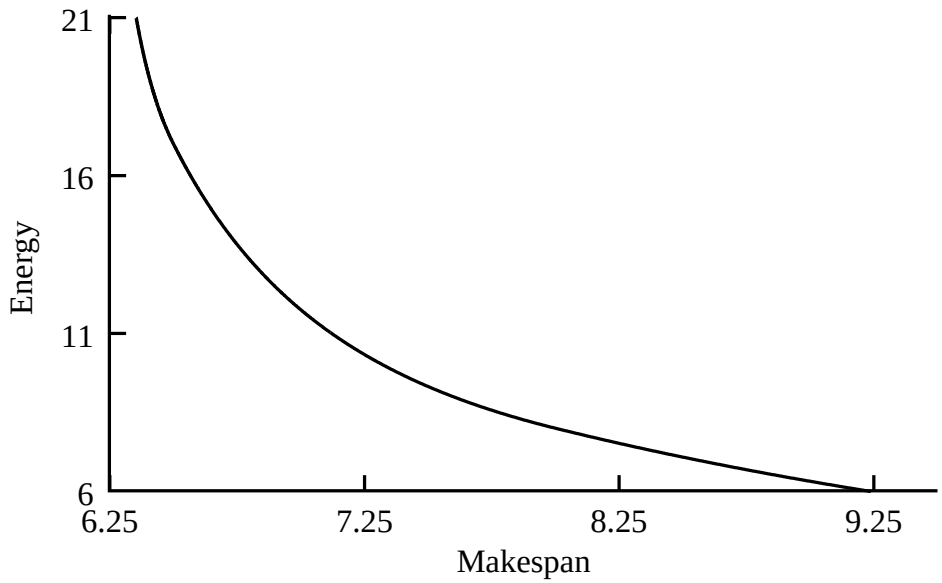

Figure 4.3: Relationship between energy and makespan in non-dominated schedules for instance with $r_1 = 0$, $\gamma_1 = 5$, $r_2 = 5$, $\gamma_2 = 2$, $r_3 = 6$, $\gamma_3 = 1$, and power = speed^3.

4.2.2 Finding all non-dominated schedules

A slight modification of $IncMerge$ finds all non-dominated schedules. Intuitively, the modified algorithm enumerates all optimal configurations (i.e. ways to break the jobs into blocks) by starting with an "infinite" energy budget and gradually lowering it. To start this process, run $IncMerge$ as above, but omit the merging step for the last job, essentially assuming the energy budget is large enough that the last job runs faster than its predecessor. To find each subsequent configuration change, calculate the energy budget at which the last two blocks merge. Until this value, only the last block changes speed. Thus, we can easily find the relationship between makespan and energy consumption for a single configuration and the curve of all non-dominated schedules is constructed by combining these. The curve for an instance with three jobs and power = speed^3 is plotted in Figure 4.3. The configuration changes occur at energy 8 and 17, but they are not readily identifiable from the figure because the makespan/energy curve is continuous and has a continuous first derivative for this power function. Higher derivatives are discontinuous at the configuration changes. Figures 4.4 and 4.5 show the first and second derivatives.

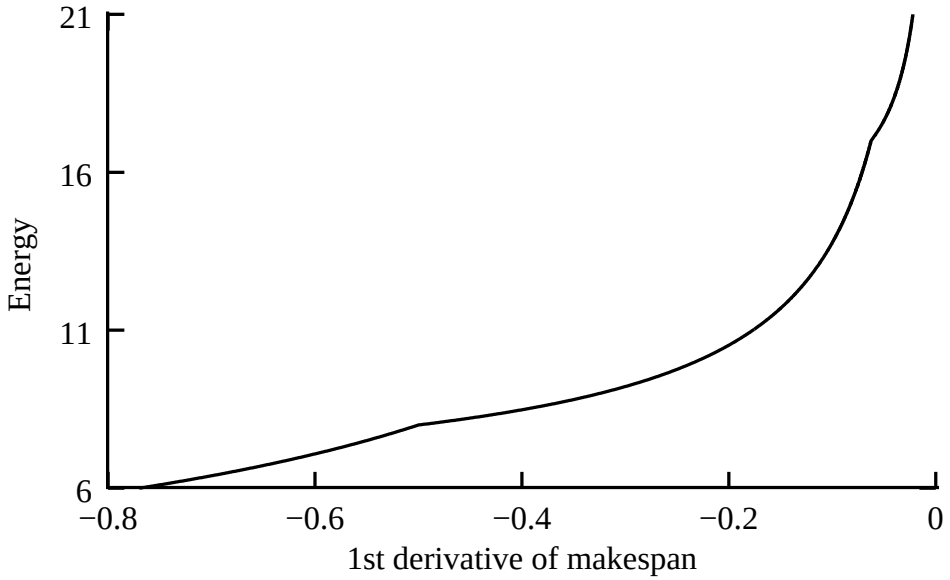

Figure 4.4: Relationship between energy and 1st derivative of makespan in non-dominated schedules for instance with $r_1 = 0$, $\gamma_1 = 5$, $r_2 = 5$, $\gamma_2 = 2$, $r_3 = 6$, $\gamma_3 = 1$, and power = speed^3.

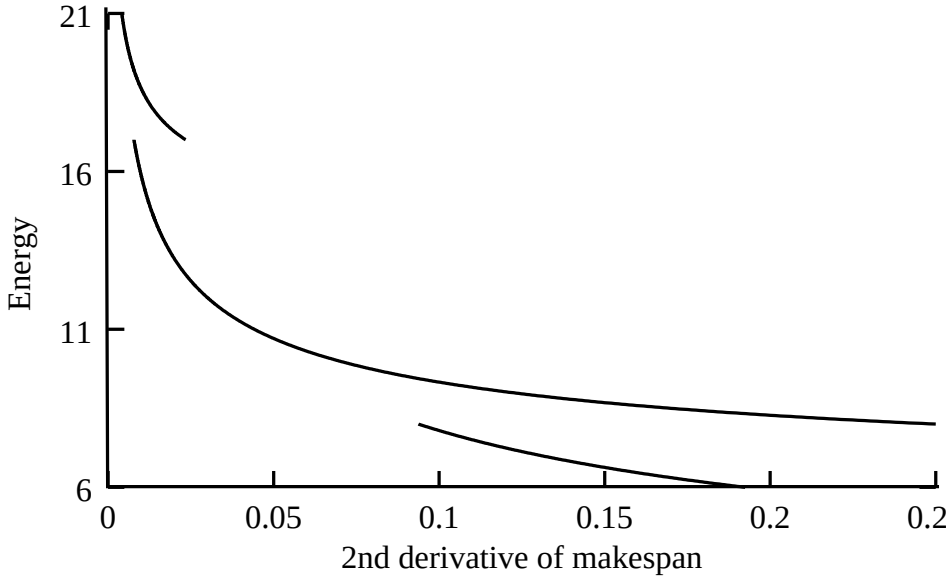

Figure 4.5: Relationship between energy and 2nd derivative of makespan in non-dominated schedules for instance with $r_1 = 0$, $\gamma_1 = 5$, $r_2 = 5$, $\gamma_2 = 2$, $r_3 = 6$, $\gamma_3 = 1$, and power = speed^3.

4.3 Impossibility of exactly minimizing total flow time

We have completely solved uniprocessor power-aware makespan by showing how to compute all non-dominated schedules, forming a curve such as Figure 4.3. We have already observed that the analogous figure from previous work on power-aware scheduling for total flow time (reproduced as Figure 4.2) was plotted with gaps where the optimal configuration involves one job completing exactly as another is released. We now show that these gaps cannot be filled exactly.

Theorem 20 *If power = speed^3, there is no exact algorithm to minimize total flow time for a given energy budget using operations $+$, $-$, $\times$, $/$, and the extraction of roots, even on a uniprocessor with equal-work jobs.*

Proof: We show that a particular instance cannot be solved exactly. Let jobs J_1 and J_2 arrive at time 0 and job J_3 arrive at time 1, each requiring one unit of work. We seek the minimum-flow schedule using 9 units of energy. Again we use σ_i to denote the speed of job J_i. Thus,

$$\sigma_1^2 + \sigma_2^2 + \sigma_3^2 = 9. \tag{4.1}$$

For energy budgets between approximately 8.43 and approximately 11.54, the optimal solution finishes job J_2 at time 1. Therefore,

$$\frac{1}{\sigma_1} + \frac{1}{\sigma_2} = 1 \tag{4.2}$$

and Theorem 13 gives us that

$$\sigma_1^3 = \sigma_2^3 + \sigma_3^3. \tag{4.3}$$

Substituting Equation (4.2) into Equations (4.1) and (4.3), followed by algebraic manipulation gives

$$\begin{aligned} 2\sigma_2^{12} - 12\sigma_2^{11} + 6\sigma_2^{10} + 108\sigma_2^9 - 159\sigma_2^8 - 738\sigma_2^7 + 2415\sigma_2^6 \\ -1026\sigma_2^5 - 5940\sigma_2^4 + 12150\sigma_2^3 - 10449\sigma_2^2 + 4374\sigma_2 - 729 \quad = \quad 0. \end{aligned}$$

According to the GAP system [45], the Galois group of this polynomial is not solvable. This implies the theorem by a standard result in Galois theory (cf. [39, pg. 542]). We owe the idea for this type

of argument to Bajaj [12]. □

Since an arbitrarily-good approximation algorithm is known for total flow time, one interpretation of Theorem 20 is that exact solutions do not have a nice representation. For most applications, the approximation is sufficient since finite precision is the normal state of affairs in computer science. Certainly, it could be used to draw an approximate curve for the gaps in the Figureflow-energy-fig. Only an exact algorithm such as $IncMerge$ can give closed-form solutions suitable for symbolic computation, however.

4.4 Multiprocessor scheduling

Now we consider power-aware scheduling of serial jobs on a multiprocessor where all the processors use a shared energy supply. This corresponds to scheduling a laptop with a multi-core processor or a server farm concerned only about total energy consumption and not the consumption of each machine separately.

In a non-dominated schedule, the processors are related by the following observations:

1. For makespan, each processor must finish its last job at the same time or slowing the processors that finish early would save energy.

2. For total flow time, each processor's last job runs at the same speed or running them at the average speed would save energy.

Using these observations, slight modifications of $IncMerge$ and the total flow time algorithm of Pruhs et al. [72] can solve multiprocessor problems once the assignment of jobs to processors is known.

We show how to assign equal-work jobs to processors for scheduling metrics with two properties. A metric is *symmetric* if it is not changed by permuting the job completion times. A metric is *non-decreasing* if it does not decrease when any job's completion time increases. Both makespan and total flow time have these properties, but some metrics do not. One example is total weighted flow time, which is not symmetric.

To prove our results, we need some notation. For schedule A and job J_i, let $\text{proc}^A(i)$ denote the index of the processor running job J_i and $\text{succ}^A(i)$ denote the index of the job run after J_i

on processor $\text{proc}^A(i)$. Also, let $\text{after}^A(i)$ denote the portion of the schedule running on processor $\text{proc}^A(i)$ after the completion of job J_i, i.e. the jobs running after job J_i together with their start and completion times. We omit the superscript when the schedule is clear from context.

We begin by observing that job start times and completion times occur in the same order.

Lemma 21 *If OPT is an optimal schedule for equal-work jobs under a symmetric non-decreasing metric, then $S_i^{OPT} < S_j^{OPT}$ implies $C_i^{OPT} \leq C_j^{OPT}$.*

Proof: Suppose to the contrary that $S_i^{OPT} < S_j^{OPT}$ and $C_i^{OPT} > C_j^{OPT}$. Clearly, jobs J_i and J_j must run on different machines. We create a new schedule OPT' from OPT. All jobs on machines other than $\text{proc}(i)$ and $\text{proc}(j)$ are scheduled exactly the same, as are those that run before jobs J_i and J_j. We set the completion time of job J_i in OPT' to C_j^{OPT} and the completion time of job J_j in OPT' to C_i^{OPT}. We also switch the suffixes of jobs following these two, i.e. run $\text{after}(i)$ on processor $\text{proc}(j)$ and run $\text{after}(j)$ on processor $\text{proc}(i)$. Job J_i still has positive processing time since $S_i^{OPT'} = S_i^{OPT} < S_j^{OPT} < C_j^{OPT} = C_i^{OPT'}$. (The processing time of job J_j increases so it is also positive.) Thus, OPT' is a valid schedule. The metric values for OPT and OPT' are the same since this change only swaps the completion times of jobs J_i and J_j.

We complete the proof by showing that OPT' uses less energy than OPT. Since the power function is strictly convex, it suffices to show that both jobs have longer processing time in OPT' than job J_j did in OPT. Job J_j ends later so its processing time is clearly longer. Job J_i also has longer processing time since runs throughout the time OPT runs job J_j, but starts earlier. □

Using Lemma 21, we prove that an optimal solution exists with the jobs distributed in *cyclic order*, i.e. job J_i runs on processor $(i \mod m) + 1$.

Theorem 22 *There is an optimal solution for equal-work jobs under any symmetric non-decreasing metric with the jobs distributed in cyclic order.*

Proof: Suppose to the contrary that no optimal schedule distributes the jobs in cyclic order. Let i be the smallest value such that no optimal schedule distributes jobs $J_1, J_2, \ldots, J_i$ in cyclic order and let OPT be an optimal schedule that distributes the first $i-1$ jobs in cyclic order. To simplify notation, we create dummy jobs $J_{-(m-1)}, J_{-(m-2)}, \ldots, J_0$, with job $J_{-(m-i)}$ assigned to processor i.

By assumption, $\text{succ}(i-m) \neq i$. Let J_l be the job such that $\text{succ}(l) = i$, i.e. the job preceeding job J_i. Since the first $i-1$ jobs are distributed in cyclic order, if we assume (without loss of generality) that jobs starting at the same time finish in order of increasing index, then Lemma 21 implies that $C_{i-m}^{OPT} \leq C_l^{OPT}$.

To complete the proof, we consider 3 cases. In each, we use OPT to create an optimal schedule assigning job J_i to processor $(i \mod m) + 1$, contradicting the definition of i.

Case 1: Suppose no job follows job J_{i-m}. We modify the schedule by moving after(l) to follow J_{i-m} on processor $(i \mod m) + 1$. Since $C_{i-m}^{OPT} \leq C_l^{OPT}$ and after(l) was able to follow job J_l, it can also follow job J_{i-m}. The resulting schedule has the same metric value and uses the same energy so it is also optimal.

Case 2: Suppose J_{i-m} is not the last job assigned to processor $\text{proc}(i-m)$ and $C_l^{OPT} < r_{\text{succ}(i-m)}$. We extend the cyclic order by swapping after(l) and after($i-m$). This does not change the amount of energy used. To show that it gives a valid schedule, we need to show that jobs J_l and J_{i-m} complete before after($i-m$) and after(l). Job J_l ends by time $S_{\text{succ}(i-m)}^{OPT}$ by the assumption that $C_l^{OPT} < r_{\text{succ}(i-m)}$. Job J_{i-m} ends by time $S_{\text{succ}(l)}^{OPT}$ since $C_{i-m}^{OPT} \leq C_l^{OPT}$.

Case 3: Suppose J_{i-m} is not the last job assigned to processor $\text{proc}(i-m)$ and $C_l^{OPT} \geq r_{\text{succ}(i-m)}$. In this case, we swap the jobs $J_{\text{succ}(i-m)}$ and $J_{\text{succ}(l)}$, but leave the schedules the same. In other words, we run job $J_{\text{succ}(i-m)}$ from time $S_{\text{succ}(l)}^{OPT}$ to time $C_{\text{succ}(l)}^{OPT}$ on processor $\text{proc}(l)$ and we run job $J_{\text{succ}(l)}$ from time $S_{\text{succ}(k)}^{OPT}$ to time $C_{\text{succ}(k)}^{OPT}$ on processor $\text{proc}(k)$. The schedules have the same metric value and each uses the same amount of energy. To show that we have created a valid schedule, we need to show that jobs $J_{\text{succ}(i-m)}$ and $J_{\text{succ}(l)}$ are each released by the start time of the other. Job $J_{\text{succ}(i-m)}$ was released by time $S_{\text{succ}(l)}^{OPT}$ since $C_l^{OPT} \geq r_{\text{succ}(i-m)}$. Job $J_{\text{succ}(l)}$ was released by time $S_{\text{succ}(i-m)}^{OPT}$ since $r_{\text{succ}(l)} \leq r_{\text{succ}(i-m)}$. (Recall that a job with index greater than i follows job J_{i-m} so $r_i = r_{\text{succ}(l)} \leq r_{\text{succ}(i-m)}$.) □

A simpler proof suffices if we specify the makespan metric since then OPT has no idle time. Thus, $r_{\text{succ}(i-m)} \leq C_{i-m}^{OPT} \leq C_l^{OPT}$ and case 2 is eliminated.

Theorem 22 allows us to solve multiprocessor makespan for equal-work jobs. Unfortunately, the general problem is NP-hard.

Theorem 23 *The laptop problem is NP-hard for nonpreemptive multiprocessor makespan, even when all jobs arrive immediately.*

Proof: We give a reduction from PARTITION [46]:

> PARTITION: Given a multiset $A = \{a_1, a_2, \ldots, a_n\}$, does there exist a partition of A into A_1 and A_2 such that $\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i$?

Let $B = \sum_{i=1}^{n} a_i$. We assume B is even since otherwise no partition exists. We create a scheduling problem from an instance of PARTITION by creating a job J_i for each a_i with $r_i = 0$ and $\gamma_i = a_i$. Then we ask whether a 2-processor schedule exists with makespan $B/2$ and a power budget allowing work B to run at speed 1.

From a partition, we can create a schedule where each processor runs the jobs corresponding to one of the A_i at speed 1. For the other direction, the convexity of the power function implies that all jobs run at speed 1 so the work must be partitioned between the processors. □

Pruhs et al. [73] observed that the special case of all jobs arriving together has a PTAS based on a load balancing algorithm of Alon et al. [5] that approximately minimizes the L_α norm of loads.

4.5 Discussion

The study of power-aware scheduling algorithms is just beginning so there are many possible directions for future work. We consider the most important to be finding online algorithms with performance guarantees for makespan or total flow time. No such algorithms are currently known, but many scheduling applications occur in the online setting. Our results on the structure of optimal solutions may help with this task, but the problem seems quite difficult. If the algorithm cannot know when the last job has arrived, it must balance the need to run quickly to minimize makespan if no other jobs arrive against the need to conserve energy in case more jobs do arrive.

Another open problem is how to handle jobs with different work requirements while minimizing multiprocessor makespan. Theorem 23 shows that finding an exact solution is NP-hard, but this does not rule out the existence of a high-quality approximation algorithm. As mentioned above,

there is a PTAS based on load balancing when all jobs arrive together. It would be interesting to see if the ideas behind this PTAS can be adapted to take release times into consideration.

We would also like to find an approximation for uniprocessor power-aware scheduling to minimize total flow time when the jobs have different work requirements. It is not hard to show that the relationships of Theorem 13 hold even in this case. Preemptions can also be incorporated with the preempted job taking the role of job J_{i+1} in the second relationship of that theorem. Thus, the problem reduces to finding the optimal configuration.

Chapter 5

Conclusions

This dissertation has given algorithms and lower bounds for problems in a number of related areas. First, we discussed admission control, giving two procedures to combine an accept-competitive algorithm and a reject-competitive algorithm into an algorithm having both types of guarantees. Then we discussed scheduling with rejections, a hybrid area that combines admission control and scheduling. Finally, we turned to power-aware scheduling, giving algorithms and hardness results for the bicriteria problems combining energy with makespan and flow.

We conclude with some high-level observations. First, in Section 5.1, we observe some connections between our work in different areas. Then, Section 5.2 discusses possible directions for future work.

5.1 Common techniques

All three problem areas considered in this dissertation fall under the general category of resource management. In addition, our results in scheduling with rejections and power-aware scheduling share some techniques.

5.1.1 Normalizing and structural properties

One shared technique we did not consciously use is identifying a normal form so there is a unique optimal schedule having this form and then determining its structure. In scheduling with rejections, the normal form required schedules to reject as few jobs as possible and run jobs in order of nondecreasing rank, with ties broken by job number. With these restrictions, we devised the offline optimal algorithm *HRF–OFF*. They also provided the framework for the online algorithms *HRF–ON* and *RR*.

In power-aware scheduling, the normal form is simply to run jobs in order of release time. We were then able to identify a list of necessary properties for an optimal schedule and prove there was a single normal schedule having these properties. Then $IncMerge$ and its multiprocessor generalization just need to find a normal schedule having these properties.

The idea of a normal form also appears in our other scheduling work, in the proof that SPT is $(P+1)/2$-competitive for total flow time [25]. That work assumes that $SRPT$ and SPT run jobs in the same order, easily achieved if ties in job length are broken with job number, and then identifies a block structure shared by these schedules.

5.1.2 Looking at special cases

The other technique that recurs in our work is focusing on special cases. We deliberately applied this technique, inspired by the quote "If there is a problem you cannot solve, there is an easier problem you can solve: find it.", attributed to Polya.[1] For scheduling with rejections, we used this technique after showing that scheduling general jobs is NP-hard and finding ourselves unable to make progress approximating it. Instead we looked at unit-length jobs, for which we were able to give an offline algorithm and some online approximations. For power-aware scheduling, we used the technique for the multiprocessor problem, restricting our attention to equal-work jobs.

5.2 Directions for future work

We conclude by discussing possible directions for future work. The last section of each chapter gives specific problems (see Sections 2.6, 3.6, and 4.5) so we focus our remarks here on more general problem areas.

5.2.1 Combining planning and scheduling

The overall goal of scheduling with rejections is to integrate planning and scheduling problems. All the previous work on these problems (including mine) has formalized this goal as trying to minimize a scheduling metric plus a term for rejection cost, but that is not the only way to approach

[1] The closest quotation I have been able to find in his writings is "If you cannot solve the proposed problem, try to solve first some related problem" [70, pg. 10].

this goal. One alternative is to look at the scheduling metric and rejection cost as two aspects of a bicriteria problem. This raises the questions "What is the best schedule achievable with rejection cost at most k?" and "What is the smallest rejection cost achievable for a schedule with metric at most k?". Even better would be finding an algorithm that gives all non-dominated solutions. Nothing is known about these questions for any scheduling metric; the existing algorithms just give a single non-dominated solution.

More generally, we believe that work on scheduling with rejections would benefit from having a specific application in mind. Only then will it be possible to determine reasonable rejection costs and how to make the tradeoff between schedule quality and rejection cost.

5.2.2 Other models for power-aware scheduling

We also believe that more interaction between theory and application would benefit theoretical power-aware scheduling. We expect to keep our emphasis on provably-good algorithms and lower bounds, but also want our work to say something about real systems. Obviously, there is a tradeoff between how much complexity is incorporated into a model and how easy it is to use that model to prove theorems, but we believe the standard power = speed^3 model is too simple. With this in mind, we have been considering ways in which the model differs from real systems that implement dynamic voltage scaling and how other features of real systems could be modeled.

The following are some of the differences:

- As discussed in Section 1.1.3, real processors supporting speed scaling typically allow the speed to be selected from a set of specific values rather than being a continuous variable. Chen et al. [32] show that scheduling jobs with deadlines in this setting is NP-hard because the existence of a maximum speed can be used to severely restrict the schedule; the reduction for 3-PARTITION creates jobs that must run at maximum speed as soon as they arrive as "dividers". This trick cannot be used without deadlines and scheduling to minimize makespan or flow, however. Our properties of optimal schedules no longer hold in the discrete setting, however, beginning with the property that each job runs at a single speed in OPT (Lemma 14).

  Two models intermediate between allowing unbounded continuous speeds and restricting

speed to values from a finite set are allowing the speed to be selected from an infinite set of discrete values (like powers of 2) or allowing it to take continuous values within some range. Qu [74] gives some experiments in the latter model.

- Another feature of real systems is that slowing down the processor has less effect on memory-bound sections of code since part of the running time is caused by memory latency. This observation has led to research in the compiler community on how to profile programs and insert speed-modifying instructions to slow the processor during memory-bound sections of code [89, 49]. That problem can be phrased by dividing jobs into segments, each with a time spent waiting for memory latency and an amount of work. The algorithm is then asked for a speed for each segment, which is the speed the processor runs throughout that segment. Freeh at al. [43] give a heuristic for this problem. Another way to phrase this problem is as scheduling jobs with different power functions, which corresponds to the problem of transmitting packets with different power functions studied by El Gamal et al. [44].

- A fairly complicated difference between our model and reality concerns battery behavior. Throughout this dissertation, we have treated batteries as stores of energy with a particular amount of energy that can be delivered in any way desired. In reality, the capacity of a battery depends on how its energy is consumed because the rate of consumption affects the chemical reactions that power the battery. Chemical batteries consist of two bars of material connected to the terminals and suspected in a liquid. The bars connected to the + and − terminals are called the *cathode* and *anode* respectively, and the liquid is called the *electrolyte*. This is illustrated in Figure 5.1. Current from the battery takes the form of electrons leaving the anode and flowing out the − terminal. This creates positive ions, which diffuse through the electrolyte. These react with the surface of the cathode, where the electrons flowing into the positive terminal create negative ions. This reaction gradually coats the cathode with a non-reactive material, eventually blocking the reaction. The energy remaining in the battery is a function of how much of the cathode has been covered.

  Lahiri et al. [59] show how battery chemistry leads to two deviations from the vision of a battery as a store for a particular amount of energy. The first deviation concerns capacity as

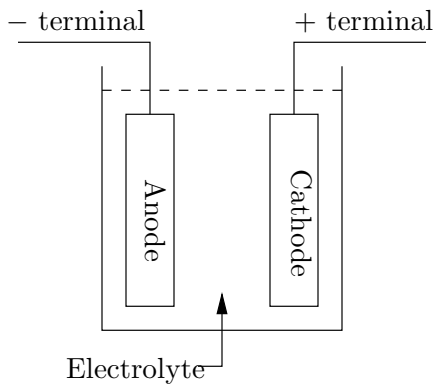

Figure 5.1: Parts of a battery

a function of the rate of power consumption. At very high rates of consumption, reactions occur unevenly on the cathode, causing the non-reactive material to deposit in a way that blocks access to some of the reaction sites. Thus, at high rates of power consumption, the battery has lower capacity. The second effect is that batteries can slightly recharge by "resting". This effect occurs because the positive ions created at the anode take time to diffuse through the electrolyte to reach the cathode. When the battery is being discharged quickly, their concentration near the cathode decreases, cutting into the reaction rate and potentially causing the battery to appear discharged even when reaction sites remain available. If the discharge rate drops for a period of time, however, the concentration of positive ions becomes uniform throughout the electrolyte, allowing the reaction to proceed.

After describing the chemical basis of these effects, Lahiri et al. [59] give a survey of work on battery modeling and how system design should be changed to incorporate these effects. Another effect, observed by Krintz et al. [57] and Wen et al. [88], is that batteries drain non-linearly, with voltage dropping faster as the battery is drained. Pedram and Wu [66] give a more thorough discussion of how to model battery performance and describe simulations to compute battery capacity under different usage profiles. They conclude that steady power consumption is the most efficient.

- A final difference between our model and real processors supporting speed scaling is that real processors must stop while their voltage is changing. Thus, they incur overhead for switching speeds. For example, Xie et al. [89] predict a transition time of 12 microseconds to go from 600MHz to 200MHz and cite published values [51] to show this corresponds well to the actual

value for Intel's XScale core. This overhead is fairly small provided that transitions are only made occasionally, but it discourages algorithms requiring frequent speed changes.

We expect to continue working on power-aware scheduling and will keep these differences in mind. Our goal is to find a model incorporating enough features of real systems to be useful, but remaining simple enough to make analysis practical.

References

[1] R. Adler and Y. Azar. Beating the logarithmic lower bound: randomized preemptive disjoint paths and call control algorithms. *J. of Scheduling*, pages 113–129, 2003. Prelimary version in Proc. 10th ACM-SIAM Symp. on Discrete Algorithms, pp. 1–10, 1999.

[2] Advanced Micro Devices, Inc. *AMD Athlon 64 processor power and thermal data sheet (ver. 3.43)*, Oct 2004. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/30430.pdf`.

[3] F.N. Afrati, E. Bampis, C. Chekuri, D.R. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Sartinutella, C. Stein, and M. Sviridenko. Approximation schemes for minimizing average weighted completion time with release dates. In *Proc. 40th Symp. Found. Computer Science*, pages 32–44, 1999.

[4] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. In *Proc. 23rd Intern. Symp. on Theoretical Aspects of Computer Science*, pages 621–633, 2006.

[5] N. Alon, Y. Azar, G.J. Woeginger, and T. Yadid. Approximation schemes for scheduling. In *Proc. 8th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 493–500, 1997.

[6] E.M. Arkin, M.A. Bender, J.S.B. Mitchell, and S.S. Skiena. The lazy bureaucrat scheduling problem. *Information and Computation*, 2002.

[7] B. Awerbuch, Y. Azar, and S. Plotkin. Throughput-competitive on-line routing. In *Proc. 34th Symp. Found. Computer Science*, pages 32–40, 1993.

[8] B. Awerbuch, Y. Bartal, A. Fiat, and A. Rosén. Competitive non-preemptive call control. In *Proc. 5th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 312–320, 1994.

[9] B. Awerbuch, R. Gawlick, T. Leighton, and Y. Rabani. On-line admission control and circuit routing for high performance computation and communication. In *Proc. 35th Symp. Found. Computer Science*, pages 412–423, 1994.

[10] Y. Azar, A. Blum, D.P. Bunde, and Y. Mansour. Combining online algorithms for acceptance and rejection. *Theory of Computing*, 1:105–117, 2005. `http://www.theoryofcomputing.org/articles/main/v001/a006/index.html`.

[11] Y. Azar, A. Blum, and Y. Mansour. Combining online algorithms for rejection and acceptance. In *Proc. 15th Annual ACM Symp. Parallelism in Algorithms and Architectures*, pages 159–163, 2003.

[12] C. Bajaj. The algebraic degree of geometric optimization problems. *Discrete Comput. Geom.*, 3:177–191, 1988.

[13] N. Bansal. *Algorithms for flow time scheduling*. PhD thesis, Carnegie Mellon University, 2003. `http://www.research.ibm.com/people/n/nikhil/papers/thesis.pdf`.

[14] N. Bansal, A. Blum, S. Chawla, and K. Dhamdhere. Scheduling for flow-time with admission control. In *Proc. 11th Annual European Symp. Algorithms*, number 2832 in LNCS, pages 43–54, 2003.

[15] N. Bansal and K. Dhamdhere. Minimizing weighted flow time. In *Proc. 14th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 508–516, 2003.

[16] N. Bansal, T. Kimbrel, and K. Pruhs. Dynamic speed scaling to manage energy and temperature. In *Proc. 45th Symp. Found. Computer Science*, pages 520–529, 2004.

[17] N. Bansal and K. Pruhs. Speed scaling to manage temperature. In *Proc. 22nd Intern. Symp. on Theoretical Aspects of Computer Science*, pages 460–471, 2005.

[18] N. Bansal, K. Pruhs, and C. Stein, 2006. Personal communication.

[19] Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, and L. Stougie. Multiprocessor scheduling with rejection. In *Proc. 7th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 95–103, 1996.

[20] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, and K.R. Pruhs. Online weighted flow time and deadline scheduling. In *Proc. 4th Intern. Workshop on Approximation Algorithms for Combinatorial Optimization*, number 2129 in LNCS, 2001.

[21] J. Błażewicz, J.K. Lenstra, and A.H.G Rinnoy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

[22] A. Blum, A. Kalai, and J. Kleinberg. Admission control to minimize rejections. *Internet Mathematics*, 1(2):165–176, 2004. Preliminary version in Proc. 7th Workshop on Algorithms and Data Structures, LNCS 2125, pp. 155–164, 2001.

[23] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P.W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[24] D.P. Bunde. Approximating total flow time. Master's thesis, Univ. Illinois at Urbana-Champaign, Dec 2002. `http://compgeom.cs.uiuc.edu/~bunde/pubs/masters.html`.

[25] D.P. Bunde. SPT is optimally competitive for uniprocessor flow. *Information Processing Letters*, 90(5):233–238, 2004.

[26] D.P. Bunde. Scheduling on a single machine to minimize total flow time with job rejections. In *Proc. 2nd Multidisciplinary Intern. Conf. on Scheduling: Theory and Applications*, pages 562–572, 2005.

[27] D.P. Bunde. Power-aware scheduling for makespan and flow. To appear in *Proc. 18th Annual ACM Symp. Parallelism in Algorithms and Architectures*, 2006.

[28] D.P. Bunde and Y. Mansour. Improved combination of online algorithms for acceptance and rejection. In *Proc. 16th Annual ACM Symp. Parallelism in Algorithms and Architectures*, pages 265–266, 2004.

[29] C. Chekuri and M.A. Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. *Journal of Algorithms*, 41:212–224, 2001.

[30] C. Chekuri and S. Khanna. Approximation schemes for preemptive weighted flow time. In *Proc. 34th ACM Symp. on Theory of Computation*, pages 297–305, 2002.

[31] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for minimizing weighted flow time. In *Proc. 33rd ACM Symp. on Theory of Computation*, pages 84–93, 2001.

[32] J.-J. Chen, T.-W. Kuo, and H.-I Lu. Power-saving scheduling for weakly dynamic voltage scaling devices. In *Proc. 9th Workshop on Algorithms and Data Structures*, number 3608 in LNCS, pages 338–349, 2005.

[33] F.A. Chudak and D.B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. In *Proc. 8th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 581–590, 1997.

[34] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in real-time systems*. John Wiley & Sons, Ltd., 2002.

[35] T. Cover and J. Thomas. *Elements of Infomation Theory*. Wiley, New York, 1991.

[36] P. Crescenzi and V. Kann. A compendium of NP optimization problems. `http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html`, 2002.

[37] M. Drozdowski. Scheduling multiprocessor tasks — an overview. *European Journal of Operational Research*, 94:215–230, 1996.

[38] J. Du, J.Y.-T. Leung, and G.H. Young. Minimizing mean flow time with release time constraints. *Theoretical Computer Science*, 75(3):347–355, 1990.

[39] D.S. Dummit and R.M. Foote. *Abstract Algebra*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[40] D. Engels, D. Karger, S. Kolliopoulos, S. Sengupta, R. Uma, and J. Wein. Techniques for scheduling with rejection. In *Proc. 6th Annual European Symp. Algorithms*, volume 1461 of *LNCS*, pages 490–501, 1998.

[41] L. Epstein and R. van Stee. Lower bounds for on-line single-machine scheduling. In *Proc. 26th Symp. Math. Found. Comput. Sci.*, number 2136 in LNCS, pages 338–350, 2001. `http://www.informatik.uni-freiburg.de/~vanstee/papers/weight.ps`.

[42] L. Epstein and R. van Stee. Optimal on-line flow time with resource augmentation. In *Proc. 13th Symp. Fund. Comp. Theory*, number 2138 in LNCS, pages 472–482. Springer-Verlag, 2001. `http://www.math.tau.ac.il/~lea/flow.ps.gz`.

[43] V.W. Freeh, F. Pan, D.K. Lowenthal, and N. Kappiah. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Proc. 10th Annual ACM Symp. Principles and Practice of Parallel Programming*, 2005.

[44] A. El Gamal, C. Nair, B. Prabhakar, E. Uysal-Biyikoglu, and S. Zahedi. Energy-efficient scheduling of packet transmissions over wireless networks. In *Proc. IEEE Infocom*, pages 1773–1782, 2002.

[45] GAP Group. Gap system for computational discrete algebra. http://turnbull.mcs.st-and.ac.uk/~gap/.

[46] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, 1979.

[47] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Math*, 5:287–326, 1979.

[48] H. Hoogeveen, M. Skutella, and G. Woeginger. Preemptive scheduling with rejection. In *Proc. 8th Annual European Symp. Algorithms*, volume 1879 of *LNCS*, pages 268–277, 2000.

[49] C.-H. Hsu. *Compiler-directed dynamic voltage and frequency scaling for CPU power and energy reduction*. PhD thesis, Rutgers University, Oct 2003. http://www.research.rutgers.edu/~chunghsu/papers/thesis.ps.

[50] P.W. Huber. Dig more coal — the PCs are coming. *Forbes*, May 1999.

[51] Intel Corporation. *Intel XScale^{TM} Core Developer's Manual*, 2002. http://developer.intel.com/design/intelxscale.

[52] S. Irani and K.R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 32(2):63–76, 2005.

[53] N. Kappiah, V.W. Freeh, and D.K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *Proc. 2005 ACM/IEEE Conf. on Supercomputing*, page 33, 2005.

[54] K. Kawamoto, J.G. Koomey, B. Nordman, R.E. Brown, M.A. Piette, M. Ting, and A.K. Meier. Electricity used by office equipment and network equipment in the u.s.: Detailed report and appendices. Technical Report LBNL-45917, Lawrence Berkeley National Laboratory, Feb 2001. http://enduse.lbl.gov/info/LBNL-45917b.pdf.

[55] H. Kellerer, T. Tautenhahn, and G.J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM J. Computing*, 28(4):1155–1166, 1999.

[56] I. Keslassy, M. Kodialam, and T.V. Lakshman. Faster algorithms for minimum-energy scheduling of wireless data transmissions. In *Proc. Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, 2003.

[57] C. Krintz, Y. Wen, and R. Wolski. Predicting program power consumption. Technical report 2002—20, University of California Santa Barbara Department of Computer Science, July 2002. http://www.cs.ucsb.edu/research/tech_reports/reports/2002-20.pdf.

[58] J. Labetoulle, E. Lawler, J. Lenstra, and A. Rinnooy Kan. Preemptive scheduling of uniform machines subject to release dates. In *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, 1984.

[59] K. Lahiri, S. Dey, D. Panigrahi, and A. Raghunathan. Battery-driven system design: A new frontier in low power design. In *Proc. of ASPDAC 2002 / VLSI Design 2002*, pages 261–267, 2002.

[60] E. Lawler, J.K. Lenstra, A. Rinnooy Kan, and D. Shmoys. Sequencing and scheduling: Algorithms and complexity. In *Logistics of production and inventory*, volume 4 of *Handbooks in operations research and management science*, chapter 9, pages 445–522. Elsevier Science, 1993.

[61] J.K. Lenstra. *Sequencing by enumerative methods*. Number 69 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam, 1977.

[62] S. Leonardi. A simpler proof of preemptive flow-time approximation. `http://www.dis.uniroma1.it/~leon/publications/srpt.ps`, 2003.

[63] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proc. 29th ACM Symp. on Theory of Computation*, pages 110–119, 1997.

[64] J.D. Mitchell-Jackson. Energy needs in an internet economy: A closer look at data centers. Master's thesis, Univ. California at Berkeley, 2001. `http://enduse.lbl.gov/Info/datacenterreport.pdf`.

[65] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.

[66] M. Pedram and Q. Wu. Design considerations for battery-powered electronics. In *Proc. 36th ACM/IEEE Design Automation Conference*, pages 861–866, 1999.

[67] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32:163–200, 2002.

[68] M.L. Pinedo. *Planning and scheduling in manufacturing and services*. Springer Series in Operations Research. Springer Science+Business Media, Inc., 2005.

[69] S. Plotkin. Competitive routing of virtual circuits in ATM networks. *IEEE J. Selected Areas in Communications*, 13(6):1128–1136, 1995.

[70] G. Polya. *How to solve it*. Princeton University Press, Princeton, NJ, 2nd edition, 1973.

[71] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In J.Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 15, pages 15–1–15–41. CRC Press, 2004.

[72] K. Pruhs, P. Uthaisombut, and G. Woeginger. Getting the best response for your erg. In *Proc. 9th Scandanavian Workshop on Algorithm Theory*, volume 3111 of *LNCS*, pages 14–25, 2004.

[73] K. Pruhs, R. van Stee, and P. Uthaisombut. Speed scaling of tasks with precedence constraints. In *Proc. 3rd Workshop on Approximation and Online Algorithms*, number 3879 in LNCS, pages 307–319, 2005.

[74] G. Qu. What is the limit of energy saving by dynamic voltage scaling? In *Proc. 2001 IEEE/ACM Intern. Conf. on Computer-Aided Design*, pages 560–563, 2001.

[75] W. Rudin. *Principles of mathematical analysis*. McGraw-Hill, Inc., 3rd edition, 1976.

[76] L. Schrage. A proof of the optimality of the shortest processing remaining time discipline. *Operations Research*, 16:687–690, 1968.

[77] S.S. Seiden. Preemptive multiprocessor scheduling with rejection. *Theoretical Computer Science*, 262:437–458, 2001.

[78] S. Sengupta. Algorithms and approximation schemes for minimum lateness/tardiness scheduling with rejection. In *Proc. 8th Workshop on Algorithms and Data Structures*, number 2748 in LNCS, pages 79–90, 2003.

[79] J. Sgall. On-line scheduling. In A. Fiat and G. Woeginger, editors, *Online Algorithms: The State of the Art*, number 1442 in LNCS, pages 196–231. Springer-Verlag, 1998.

[80] D. Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26(1):197–199, 1976.

[81] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing power in high-performance microprocessors. In *Proc. 35th ACM/IEEE Design Automation Conference*, pages 732–737, 1998.

[82] Trubador. How to fry an egg on an XP. `http://www.phys.ncku.edu.tw/~htsu/humor/fry_egg.html`.

[83] E. Uysal-Biyikoglu and A. El Gamal. On adaptive transmission for energy efficiency in wireless data networks. *IEEE Trans. on Information Theory*, 50(12):3081–3094, Dec. 2004.

[84] E. Uysal-Biyikoglu, B. Prabhakar, and A. El Gamal. Energy-efficient packet transmission over a wireless link. *IEEE/ACM Trans. Networking*, 10(4):487–499, Aug. 2002.

[85] B. Veltman, B.J. Lageweg, and J.K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.

[86] A. Vestjens. *On-line scheduling*. PhD thesis, Eindhoven University of Technology, Nov 1997. `ftp.win.tue.nl/pub/techreports/vestjens/PhDthesis.ps`.

[87] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. 1st Symp. on Operating Systems Design and Implementation*, pages 13–23, 1994.

[88] Y. Wen, R. Wolski, and C. Krintz. Online prediction of battery lifetime for embedded and mobile devices. In *Proc. 3rd Intern. Workshop Power-Aware Computer Systems*, number 3164 in LNCS, pages 57–72, 2004.

[89] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proc. 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 49–62, 2003.

[90] A. Yao. Towards a unified measure of complexity. In *Proc. 18th Symp. Found. Computer Science*, pages 222–227, 1977.

[91] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proc. 36th Symp. Found. Computer Science*, pages 374–382, 1995.

Appendix A

Glossary of terms

This appendix gives a short definition for many of the terms used in the dissertation. For the meaning of scheduling notation, refer to Table 1.3 on page 12.

accept-competitive The competitive ratio of an admission control algorithm when the metric is the number (or value) of requests accepted. For example, an algorithm is 2-accept-competitive if it always accepts at least half as many requests as the optimal algorithm.

active job A job that has been released, but not yet completed.

admission control A type of resource management problem in which the algorithm has a limited pool of resources and must select a subset from an incoming sequence of requests that can be satisfied.

anode The material in a battery connected to the negative terminal.

approximation algorithm An algorithm whose performance is near optimal in the following sense: If the objective is to minimize the metric, then the algorithm achieves a value at most some multiple of the optimal solution's value. If the objective is to maximize the metric, then the algorithm achieves a least some fraction of the optimal solution's value. For example, a 2-approximation gives a solution whose value is no more than twice as much for a minimization problem and at least one half as much for a maximization problem. Used when no fast algorithm that solves the problem exactly is known.

approximation ratio The value giving the quality of an approximation algorithm. For example, a 2-approximation has approximation ratio 2.

benefit problem The version of an optimization problem with the objective of minimizing cost. The other version, with the objective of maximizing benefit, is called the benefit problem.

block The analysis of scheduling algorithms often divides the jobs into blocks, the definition of which varies by context. In Chapter 4, it is used to mean a maximal substring of scheduled jobs such that each job except the last finishes after the arrival of its successor.

busy A type of schedule in which the processor is never idle unnecessarily; it will always run a job if one is available. Also used to describe algorithms that always give busy schedules.

call control A special case of admission control in which the resource being managed is bandwidth in a network and requests are communication paths called calls.

capacity For call control problems, the capacity of an edge is the number of requests that can use that edge.

cathode The material in a battery connected to the positive terminal.

competitive analysis A concept similar to approximation, but applied to online algorithms. For example, a 2-competitive online algorithm gives a solution whose value is at most twice optimal for a minimization problem or at least one half as much for a maximization problem.

competitive ratio The value giving the quality of an algorithm according to competitive analysis. For example, a 2-competitive algorithm ha competitive ratio 2.

completion time The completion time of a job is the time at which it has exclusively occupied the processor for an amount of time equal to its processing time. When used to refer to a schedule, it denotes the metric total completion time, which is equal to the sum of completion times of each job. The completion time of job J_i scheduled by algorithm A is denoted C_i^A.

cost problem The version of an optimization problem with the objective of maximizing benefit. The other version, with the objective of minimizing cost, is called the cost problem.

cyclic order A distribution of jobs to m processors defined in Chapter 4 in which job J_i runs on processor $(i \mod m) + 1$.

deadline A job parameter giving the time at which the job should (or must) be completed. The deadline of job J_i is denoted d_i.

dynamic voltage scaling Another name for speed scaling specifically when the adjustable value is processor speed. Comes from the fact that a slower processor can be run at lower voltage.

electrolyte The liquid part of a battery between the anode and the cathode.

encoding scheme Used for sending transmissions in the presence of interference. The encoding scheme tells the transmitter what to send for each possible message. Also called an error correcting code.

Euler's constant The limit as n approaches infinity of $H_n - \ln n$, where $H_n = \sum_{i=1}^{n} 1/i$ is the n^{th} Harmonic number. Denoted γ and having value approximately 0.577. See Rudin [75] for more information.

feasible In admission control, a feasible set of requests is one whose requests can all be simultaneously satisfied.

flow The flow of a job is its completion time minus its release time, i.e. the time is spent in the system. As a metric, refers to the sum of job flows. The flow of job J_i when scheduled by algorithm A is denoted F_i^A.

flow shop scheduling A scheduling model in which jobs must be run on multiple processors, representing different stations, in a particular order. See also open shop scheduling.

FPTAS see PTAS

handled In scheduling with rejections, we say a job is handled when it is either rejected or completed.

handling time The analog of completion time for scheduling with rejections; the time when a job is either rejected or completed. Also denoted C_i^A and used instead of completion time to derive metrics like flow.

Harmonic number The n^{th} Harmonic number is $\sum_{i=1}^{n} 1/i$.

identical processors A multiprocessor setting where all processors are equivalent.

idle time The time a job is active but not assigned to a processor. The idle time of a schedule is the sum of the job idle times. Also called waiting time.

laptop problem A special case of power-aware scheduling with the objective of maximizing schedule quality while operating with a specific energy budget.

lateness The lateness of a job is is the amount after its deadline that it completes. If the job finished before its deadline, its lateness is negative. As a metric, lateness refers to the maximum job lateness. The lateness of job J_i when scheduled by algorithm A is denoted by L_i^A.

lazy rejection As used in Chapter 2, the attempt to delay rejections as long as possible when combining simulations of two admission control algorithms. When one of the simulated algorithms wants to reject a job, that job is instead marked. Marked jobs are only rejected when the set of jobs is not feasible with them.

locally competitive A concept only meaningful for metrics whose value increase over time, like flow. An algorithm is locally c-competitive if its value's rate of increase is always at most c times as much as optimal. For flow and weighted flow, this is equivalent to being c-competitive [20].

makespan A scheduling metric equal to the latest job completion time.

maximization problem An optimization problem where the goal is to maximize the objective.

minimization problem An optimization problem where the goal is to minimize the objective.

multiprocessor A scheduling problem in which several processors must be scheduled. If the processors are identical, the processing time of a job is unaffected by which processor it is assigned to. If the processors are uniform, then each processor has a speed and job processing times vary linearly with speed. Finally, if the processors are unrelated, then the processing time of each job is an unrestricted function of the processor to which it is assigned. This this proposal, we assume the processors are identical. Jobs in the multiprocessor setting can be serial or parallel. Related models are open shop scheduling and flow shop scheduling.

non-decreasing A non-decreasing scheduling metric is one that does not decrease when any job's completion time increases.

non-dominated solution A solution to a bicriteria problem such that any other solution at least as good as it in terms of one of the objectives is inferior in terms of the other objective.

nonpreemptive Scheduling setting where jobs must be run to completion without interruptions once started.

normalized optimal solution As used in Chapter 3, an optimal solution that rejects the fewest number of jobs and always runs the active job with highest rank, breaking ties with the job number.

offline A problem in which all input is available before the algorithm makes any decisions.

online A problem in which the input arrives over time and the algorithm makes decisions based only on the input so far.

open shop scheduling A scheduling model in which jobs must be run on multiple processors, representing different stations, where the runs can occur in any order. See also flow shop scheduling.

OPT Used to denote the optimal algorithm or its solution.

parallel job A job that must be assigned to several processors simultaneously. One type of job in parallel scheduling problems. (Serial jobs run on a single processor.) Special types of parallel jobs are malleable and moldable.

power Energy per unit time. Used for power-aware scheduling.

preemption In a scheduling setting, the act of pausing a job, which can later be resumed without penalty. For admission control, the act of rejecting a job that had previously been accepted.

preemptive A type of algorithm using preemption.

pseudoscheudle An assignment of jobs to processors and times in a way that is not necessarily a legal schedule.

PTAS An acronym for Polynomial-Time Approximation Scheme. This is an algorithm that is a $(1+\epsilon)$-approximation for any value of $\epsilon > 0$ where the running time is polynomial in n for any fixed ϵ. A special type is the Fully-Polynomial-Time Approximation Scheme (FPTAS), whose running time is polynomial in both n and $1/\epsilon$.

rate When talking about an encoding scheme, the rate is the number of bits encoded per message.

reject-competitive The competitive ratio of an admission control algorithm when the metric is the number (or value) of requests rejected. For example, an algorithm is 2-reject-competitive if it always rejects no more than twice as many requests as the optimal algorithm.

rejection cost The penalty for rejecting a job in scheduling problems where this is possible. The rejection cost of job J_i is denoted c_i. When all rejection costs are the same, this value is denoted c.

release time The time at which is job becomes available to run. In online problems, this is the time at which the algorithm becomes aware of the job. The release time of job J_i is denoted r_i.

restart Scheduling setting where jobs can be stopped before completion, but must then be run from the beginning. Intermediate in power between the preemptive and nonpreemptive settings.

scheduling A type of resource management problem in which the algorithm must decide when to run each of a list of incoming jobs.

scheduling with rejections A combination of scheduling and admission control. The algorithm must either reject or schedule each job.

serial job A job that runs on a single processor. One type of job in parallel scheduling problems. (Parallel jobs must be assigned to several processors simultaneously.)

server problem A special case of power-aware scheduling with the objective of minimizing energy consumption while achieving a specific schedule quality.

speed scaling The ability to conserve energy by operating more slowly. Examples discussed in this dissertation include processors that can run at different frequencies (also called dynamic

voltage scaling) and transmitters that can use lower power if they switch to an encoding scheme with a lower rate.

SPT An acronym for Shortest Processing Time. It is the nonpreemptive scheduling algorithm that starts the job with least processing time whenever a processor becomes free.

SRPT An acronym for Shortest Remaining Processing Time. It is the preemptive scheduling algorithm that always runs the job(s) with least remaining processing time. Known to be optimal for sum of completion times and total flow in the uniprocessor setting [76, 80].

stream A common element in instances for lower bound constructions. A stream consists of a sequence of short jobs arriving as quickly as they can be processed on a processor that does nothing else. For example, a stream of k unit-length jobs starting at time t arrive at times $t, t+1, t+2, \ldots, t+k-1$.

SWITCH Procedure by Azar, Blum, and Mansour [11] to combine a c_A-accept-competitive algorithm and a c_R-reject-competitive algorithm into a single algorithm that is simultaneously $O(c_A^2)$-accept-competitive and $O(c_A c_R)$-reject-competitive.

symmetric A symmetric scheduling metric is one whose value is unchanged if the job completion times are permuted.

tardiness The amount a job finishes after its deadline, but defined to be 0 if the job finishes before its deadline. A non-negative version of lateness. As a metric, tardiness refers to the maximum job tardiness. The tardiness of job J_i when scheduled by algorithm A is denoted by T_i^A.

throughput In call control, this refers to the amount of link capacity that is used, the sum of each accepted request's duration and bandwidth requirement. Also used in scheduling, but with various meanings.

total completion time Scheduling metric whose value is the sum of job completion times, i.e. $\sum_i C_i^A$.

total flow time Scheduling metric whose value is the sum of each job's flow, the time between its release and completion. In other words, $\sum_i F_i^A = \sum_i (C_i^A - r_i)$.

total weighted completion time Scheduling metric similar to total completion time except that the completion time of each job is multiplied by its weight. In other words, $\sum_i w_i C_i^A$.

total weighted flow time Scheduling metric similar to total flow time except that the flow of each job is multiplied by its weight. In other words, $\sum_i w_i F_i^A = \sum_i w_i (C_i^A - r_i)$.

uniform-length jobs A special case of scheduling problems when all jobs have the same processing time. Slightly more general than unit-length jobs.

uniform processors A multiprocessor setting where some processors can be faster than others, but the relative speed is independent of the job being run.

unrelated processors A multiprocessor setting where the completion time of a job is an arbitrary function of the processor running it.

uniprocessor A scheduling problem in which only a single processor is scheduled.

unit-length jobs A special case of scheduling problems when all jobs have the same processing time and each job arrival time can be expressed as a multiple of this value. We normalize this length to one so that job arrival and processing times are restricted to be integers.

waiting time See idle time.

weight A job parameter identifying the relative importance of this job. The weight of job J_i is denoted w_i.

weighted completion time A version of the completion time metric in which the completion time of each job is multiplied by its weight before being summed.

weighted flow The weighted flow of a job is its flow (completion time minus release time) multiplied by a job-specific weight. As a metric, refers to the sum of job weighted flow.

work The measure of job size and processor accomplishment in power-aware scheduling. The work done by a processor is the integral over time of the processor speed.

work requirement In power-aware scheduling, the amount of work that must be done on a job in order to complete it. The work requirement of job J_i is denoted γ_i.

Appendix B

Proof of Lemmas 3 and 4

This appendix gives proofs for the following properties of $\tau_{k,k}$, originally stated in Chapter 3:

Lemma 3 (restated) *For all values of k, $\tau_{k,k} < 1$.*

Lemma 4 (restated) $\lim_{k\to\infty} \tau_{k,k} = 1$.

Before proving these results, we show the following technical lemma:

Lemma 24 *Consider the axis-aligned rectangle with corners $(i, 1/i)$ and $(i+1, 1/(i+1))$. The proportion of its area above the function $f(x) = 1/x$ is at most $1/2 + 1/(2i)$.*

Proof: The area of the box is $1/i - 1/(i+1) = 1/(i(i+1))$. The area within the box and under the curve is

$$\int_i^{i+1} \frac{dx}{x} - \frac{1}{i+1} = \ln\frac{i+1}{i} - \frac{1}{i+1}$$

Thus, the proportion of area under the curve is $1 - i(i+1)\ln((i+1)/i) + i = 1 + i - i(i+1)\ln(1+1/i)$. Using the first two terms of the Taylor series expansion for $\ln(1+x)$, this is at most

$$1 + i - (i^2 + i)\left(\frac{1}{i} - \frac{1}{2i^2}\right) = 1/2 + 1/(2i)$$

□

This allows us to prove the first of our lemmas about $\tau_{k,k}$.

Proof of Lemma 3: For the value of $\tau_{k,k}$, we use Equation 3.2:

$$\tau_{k,k} = \sum_{j=1}^{k} \frac{1}{(k+1)/(e-1) + j - 1} = \int_1^{k+1} \frac{dx}{(k+1)/(e-1) + \lfloor x \rfloor - 1}$$

We want to remove the floor from this integral, but doing so decreases the value. We bound the difference by considering the total area in the axis-aligned rectangles with corners along the curve

$y = 1/x$ at $x = (k+1)/(e-1) + i$ for $i = 0, 1, \ldots, k$. These boxes have total area

$$\sum_{i=(k+1)/(e-1)}^{(k+1)/(e-1)+k} \left(\frac{1}{i} - \frac{1}{i+1} \right) = \frac{e-1}{k+1} - \frac{e-1}{ke+1}$$

When $k \geq 10$, the proportion of this area that is part of the error is at most

$$\frac{1}{2} + \frac{1}{2i} \leq \frac{1}{2} + \frac{e-1}{22} = \frac{e+10}{22} \leq \frac{1}{e-1}$$

by Lemma 24. Thus, $\tau_{k,k}$ for $k \geq 10$ obeys

$$\begin{aligned}
\tau_{k,k} &\leq \int_1^{k+1} \frac{dx}{(k+1)/(e-1) + x - 1} + \frac{1}{e-1} \left(\frac{e-1}{k+1} - \frac{e-1}{ke+1} \right) \\
&= \int_1^{k+1} \frac{(e-1)dx}{k+2-e+x(e-1)} + \frac{k(e-1)}{(k+1)(ke+1)} \\
&< \ln(k+2-e+(k+1)(e-1)) - \ln(k+2-e+(e-1)) + \frac{e-1}{ke+1} \\
&= \ln \frac{ke+1}{k+1} + \frac{e-1}{ke+1} = 1 - \ln \frac{ke+e}{ke+1} + \frac{e-1}{ke+1} \\
&= 1 - \ln \left(1 + \frac{e-1}{ke+1} \right) + \frac{e-1}{ke+1}
\end{aligned}$$

The last line is at most 1 since $\ln(1+x) = \sum_{i=1}^{\infty} (-1)^{i+1} x^i / i$, which is greater than x for $x < 1$.

For $k < 10$, we verify the identity by hand using Equation 3.2: $\tau_{1,1} \approx 0.859$, $\tau_{2,2} \approx 0.937$, $\tau_{3,3} \approx 0.961$, $\tau_{4,4} \approx 0.972$, $\tau_{5,5} \approx 0.979$, $\tau_{6,6} \approx 0.983$, $\tau_{7,7} \approx 0.985$, $\tau_{8,8} \approx 0.987$, $\tau_{9,9} \approx 0.989$. □

The other observation follows quickly from this:

Proof of Lemma 4:

$$\begin{aligned}
\tau_{k,k} &= \sum_{j=1}^{k} \frac{1}{(k+1)/(e-1) + j - 1} \\
&\geq \sum_{j=1}^{k} \frac{1}{\lceil (k+1)/(e-1) \rceil + j - 1} \\
&= H_{\lceil (k+1)/(e-1) \rceil + k - 1} - H_{\lceil (k+1)/(e-1) \rceil - 1}
\end{aligned}$$

where $H_n = \sum_{j=1}^{n} 1/n$ is the n^{th} *Harmonic number*. Recall that $\lim_{n \to \infty} (H_n - \ln n) = \gamma$, where

$\gamma \approx 0.577$ is *Euler's constant* [75]. Thus,

$$\begin{aligned}
\lim_{k\to\infty} \tau_{k,k} &\geq \lim_{k\to\infty} \left(H_{\lceil (k+1)/(e-1)\rceil + k - 1} - H_{\lceil (k+1)/(e-1)\rceil - 1}\right) \\
&= \lim_{k\to\infty} \left(\ln((k+1)/(e-1) + k - 1) - \ln((k+1)/(e-1) - 1)\right) \\
&= \lim_{k\to\infty} \ln \frac{(k+1)/(e-1) + k - 1}{(k+1)/(e-1) - 1} = \ln(1 + e - 1) \\
&= 1
\end{aligned}$$

The result then follows from Lemma 3. □

Curriculum Vitæ

David P. Bunde

Department of Computer Science

University of Illinois at Urbana-Champaign

Research Interests

Scheduling, processor allocation in high-performance computers, online algorithms, algorithmic questions in graph theory.

Education

1998–present **University of Illinois at Urbana-Champaign**:

Ph.D. in Computer Science, October 2006 (expected). Advisor: Jeff Erickson.

Thesis: Scheduling and Admission Control

M.S. in Computer Science, December 2002. Advisor: Jeff Erickson.

Thesis: Approximating Total Flow Time

1994–1998 **Harvey Mudd College**: B.S. Double major in Mathematics and Computer Science, May 1998.

Employment

Since 1998 **University of Illinois at Urbana-Champaign Department of Computer Science.** Graduate student research assistant or teaching assistant during academic years. Instructor during summer 2004 and summer 2005.

2000–2003 **Sandia National Labs**, Albuquerque, NM. Intern during summers. Worked on research projects in processor allocation, scheduling, and robot construction.

Teaching (all at UIUC)

Spring 2006 Co-taught new version of CS 273: Introduction to Theory of Computation with faculty. Helped design and teach new course on formal languages. Responsibilities included choosing material, lecturing, designing assignments, holding office hours, and grading.

Summer 2005 Co-taught CS273: Introduction to Theory of Computation with another graduate student. Responsibilities included lecturing, designing assignments, holding office hours, and grading.

Summer 2004 Co-taught CS273: Introduction to Theory of Computation with another graduate student. Responsibilities included lecturing, designing assignments, holding office hours, and grading.

Fall 2000 Teaching assistant for CS 375: Automata, Formal Languages, Computational Complexity. (Now CS 475.) Responsibilities included designing assignments, holding office hours, grading, and giving some lectures.

Spring 2000 Teaching assistant for CS 273: Introduction to Theory of Computation. Responsibilities included designing assignments, holding office hours, and grading.

Fall 1999 Teaching assistant for CS 373: Combinatorial Algorithms. (Now CS 473.) Responsibilities included designing assignments, holding office hours, and grading.

Spring 1999 Teaching assistant for CS 225: Data Structures and Software Principles. Responsibilities included teaching lab sections, designing assignments, holding office hours, and grading.

Fall 1998 Teaching assistant for CS 225: Data Structures and Software Principles. Responsibilities included teaching lab sections, designing assignments, holding office hours, and grading.

I was listed in the University's "Incomplete List of Teachers Ranked as Excellent by Their Students" based on my teaching evaluations earned for Fall 1998 and Spring 1999.

Invited Lectures

2006 Cal Poly, San Luis Obispo, CA; Earlham College, Richmond, ID; Colgate University, Hamilton, NY; Center for Computing Sciences, Bowie, MD; Knox College, Galesburg, IL; Trinity College, Hartford, CT; Mt. Holyoke College, South Hadley, MA; Louisiana State University, Baton Rouge, LA; Indiana State University at South Bend.

Conference and Workshop Presentations

2006 Workshop on Scheduling Algorithms for New Emerging Applications. Luminy, France.

2005 9th Workshop on Algorithms and Data Structures. Waterloo, Canada.
2nd Multidisciplinary International Conference on Scheduling: Theory & Applications. New York, NY.

2004 16th ACM Symposium on Parallelism in Algorithms and Architectures. Barcelona, Spain.
3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems. Santa Fe, NM.

2002 4th IEEE International Conference on Cluster Computing. Chicago, IL.

Publications

Copies of these papers can be downloaded from http://`compgeom.cs.uiuc.edu/~bunde/pubs`. Each paper is listed once, even if it has appeared in multiple versions.

Refereed Journal Papers

[1] Combining Online Algorithms for Acceptance and Rejection. Written with Y. Azar, A. Blum, and Y. Mansour. *Theory of Computing* 1:105-117, 2005. Combines and improves two confer-

ence papers. A preliminary version of my contributions (written with Y. Mansour) appeared in *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–266, 2004.

[2] SPT is optimally competitive for uniprocessor flow. *Information Processing Letters* 90(5):233–238, 2004.

Refereed Conference Papers

[3] Power-Aware Scheduling for Makespan and Flow. To appear in *Proceedings of the 18th Annual ACM Symp. Parallelism in Algorithms and Architectures*, 2006. (arXiv:cs.DS/0605126)

[4] Communication-aware processor allocation for supercomputers. Written with M.A. Bender, E.D. Demaine, S.P. Fekete, V.J. Leung, H. Meijer, and C.A. Phillips. *Proceedings of the 9th Workshop on Algorithms and Data Structures*, volume 3608 of LNCS, pages 169–181, 2005. (arXiv:cs.DS/0407058) Invited and submitted to the special issue of *Algorithmica* devoted to this conference.

[5] Scheduling on a single machine to minimize total flow time with job rejections. *Proceedings of the 2nd Multidisciplinary International Conference on Scheduling: Theory & Applications*, pages 562–572, 2005.

[6] Communication patterns and allocation strategies. Written with V.J. Leung and J. Mache. *Proceedings of the 3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, 2004.

[7] Processor allocation on Cplant: Achieving general processor locality using one-dimensional allocation strategies. Written with V. Leung, E. Arkin, M. Bender, J. Johnston, A. Lal, J. Mitchell, C. Phillips, and S. Seiden. *Proceedings of the 4th IEEE International Conference on Cluster Computing*, pages 296–304, 2002.

Other Papers (including unpublished work)

[8] Parity edge-coloring of graphs. Written with K. Milans, D.B. West, and H. Wu. Submitted, 2006. (arXiv:math.CO/0602341)

[9] Distance-2 Edge Coloring is NP-Complete. Written with J. Erickson and S. Thite. Unpublished note, 2005. (arXiv:cs.DM/0509100)

[10] Pebbling and Optimal Pebbling in Graphs. Written with E.W. Chambers, D. Cranston, K. Milans, and D.B. West. Submitted, 2005. (arXiv:math.CO/0510621)

[11] *Approximating Total Flow Time*. Masters thesis, Computer Science Department, University of Illinois at Urbana-Champaign, December 2002.

Service

- Referee for *ACM Computing Surveys*, *IEEE Transactions on Parallel and Distributed Systems*, *Information and Computation*, and *Operations Research Letters*.
- External reviewer for ACM Symposium on Parallelism in Algorithms and Architectures (2006); ACM Symposium on Computational Geometry (2006); Europar (2005); ACM-SIAM Symposium on Discrete Algorithms (2002, 2003, 2006); Symposium on Theory of Computation (2005); Scandinavian Workshop on Algorithm Theory (2004); and International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (2004).
- Organizer of Midwest Theory Day (informal 1-day theoretical computer science meeting), Spring 2005.
- Graduate student member of Fellowships, Assistantships, and Admissions Committee during 2003–2004 academic year.
- Advisor for undergraduate students (hired by CS department), 1998–2003.
- Coordinator for CS Grad Students Organization during 2000–2001 and 2001–2002 academic years.
- Organizer of CS theory seminar at University of Illinois at Urbana-Champaign, 2001–2003.
- Volunteer for *Theory of Computing*, preparing bibliographies for publication
- Member of ACM and IEEE.