Combinatorial algorithms for packing and scheduling problems

Habilitationsschrift Universität Karlsruhe 2006

Rob van Stee

Contents

1 Introduction **1**
1.1 Approximation algorithms 2
1.2 On-line algorithms 2
1.3 Multidimensional packing 4
1.4 Scheduling 6
1.5 Outline of the thesis 8
1.6 Credits 9

I Multidimensional packing **13**

2 Multidimensional packing problems: a survey **15**
2.1 Next Fit Decreasing Height 16
2.2 Strip packing 18
2.2.1 Online results 18
2.2.2 Offline results 18
2.2.3 Rotations 20
2.3 Two-dimensional bin packing 20
2.3.1 Online results 20
2.3.2 Offline results 22
2.3.3 Resource augmentation 23
2.3.4 Rotations 23
2.4 Column (three-dimensional strip) packing 23
2.4.1 Online and offline results 23
2.4.2 Rotations 24
2.5 Three- and more-dimensional bin packing 24
2.6 Vector packing 25
2.7 Variations 27
2.7.1 Rectangle stretching 27
2.7.2 Items appear from the top 27
2.7.3 Dynamic bin packing 28
2.7.4 Packing rectangles in a single rectangle 28

3 An approximation algorithm for square packing 31
3.1 Subroutines for the algorithm . . . 31
3.2 Algorithm . . . 33
3.3 Approximation ratio . . . 34

4 Optimal online algorithms for multidimensional packing 37
4.1 Packing hypercubes . . . 38
4.2 Packing hyperboxes . . . 44
4.3 Variable-sized packing . . . 49
4.4 Resource augmented packing . . . 53
4.4.1 The asymptotic performance ratio . . . 53
4.5 Conclusions . . . 55

5 Packing with rotations 57
5.1 Strip packing . . . 58
5.1.1 A $3/2$-approximation algorithm . . . 59
5.1.2 An asymptotic polynomial-time approximation scheme . . . 61
5.2 Two-dimensional bin packing . . . 67
5.3 This side up . . . 74
5.4 Further applications . . . 77
5.4.1 Three-dimensional strip packing . . . 77
5.4.2 Three-dimensional bin packing . . . 79
5.5 Conclusion . . . 80

II Scheduling 81

6 Minimizing the total completion time online on a single machine, using restarts 83
6.1 Algorithm RSPT . . . 84
6.2 Global assumptions and event assumptions . . . 86
6.3 Definitions and notation . . . 88
6.4 Amortized analysis . . . 90
6.4.1 Credit requirements . . . 92
6.4.2 The invariant . . . 93
6.4.3 Analysis of an event . . . 96
6.5 Interruptions . . . 98
6.6 Job completions . . . 106
6.6.1 OPT runs at least three jobs before *ARRIVE* . . . 109
6.7 Interruptions, $s < x/2$ . . . 111

7 Online scheduling of splittable tasks 117
7.1 A greedy algorithm . . . 118
7.2 Computing the optimal makespan . . . 119
7.2.1 Offline algorithm for $\ell \geq (m+1)/2$ . . . 120

7.3 Algorithm HIGH$(k, \mathcal{R})$ 120
7.3.1 Many splits 121
7.3.2 The case of $f = m - 1$ fast machines 123
7.3.3 Few splits on identical machines 126
7.4 A special case: four machines, two parts 127
7.5 Conclusion 131

8 Speed scaling of tasks with precedence constraints 133
8.1 Motivation 133
8.2 Summary of results 134
8.2.1 Related results 135
8.3 Formal problem description 135
8.4 No precedence constraints 136
8.5 Main results 137
8.5.1 One speed for all machines 137
8.5.2 The power equality 137
8.5.3 Algorithm 142
8.5.4 Analysis 144

9 Real-time integrated prefetching and caching 147
9.1 Problem definition 148
9.2 Problem properties 149
9.3 Algorithm REALMISER 149
9.3.1 Analysis of REALMISER 151
9.4 Online algorithms 156
9.5 Conclusions 158

Chapter 1

Introduction

This thesis is concerned with various optimization problems in the field of packing and scheduling. We will develop algorithms for these problems that are guaranteed to give solutions that are not too far away from the optimal solution. There are two main approaches developing such algorithms. The first is *LP-based* algorithms, where the problem is first modeled as a linear program (LP) by relaxing some of the constraints. The solution of this linear program is then rounded to find a feasible solution to the original problem, and we compare this solution to the optimal solution to get a performance guarantee for the algorithm.

The second approach is *combinatorial* algorithms. This is the approach that we will focus on in this thesis. Here, we analyze the combinatorial structure of a problem in order to find general rules that can be proven to give good results for all problem instances.

In this thesis we will consider both approximation algorithms (Section 1.1), that are given a complete problem instance but need to generate a solution in polynomial time, and online algorithms (Section 1.2). Online algorithms receive their input incrementally and need to make decisions without knowing the rest of the input. Such algorithms are required in situations where solutions need to be generated over time and the input is only completely known at the end of processing.

The first part of our thesis is concerned with multidimensional bin packing. Bin packing is one of the oldest and most well-studied problems in computer science [42, 53]. The study of this problem dates back to the early 1970's, when computer science was still in its formative phase—ideas which originated in the study of the bin packing problem have helped shape computer science as we know it today. The influence and importance of this problem are witnessed by the fact that it has spawned off whole areas of research, including the fields of online algorithms and approximation algorithms. We introduce this part in Section 1.3.

The second part of our thesis deals with another classical area in computer science: scheduling. This area has been studied intensively since the 1960's [83] and many different problem settings have been studied, for instance machines with different speeds (related machines) or with availability constraints [149], jobs that have precedence constraints, jobs that need to be executed in parallel, jobs that cannot be executed in parallel, jobs with different stages and many, many other settings. We introduce this problem area in Section 1.4.

An outline of the results described in this thesis can be found in section 1.5.

1.1 Approximation algorithms

It can be shown for many important problems that determining an optimal solution may be extremely time-consuming due to their computational complexity. The class of NP-complete problems represents a large collection of such problems, which are all related in the sense that a polynomial-time solution of one of them implies the polynomial-time solvability of the whole class. Up to now, no polynomial-time algorithm for an NP-complete problem is known. For more background about the complexity classes P and NP, see for instance Garey and Johnson [80].

Example Given an input of n jobs of different sizes, assign them to m machines such that the maximum load is minimized, where the load of a machine is the total size of the jobs assigned to it. This problem is strongly NP-hard.

For such problems, we can try to find relatively simple algorithms that are guaranteed to find "nearly" optimal solutions. An *approximation algorithm* should have a polynomial running time and produce a feasible solution with cost at most some factor away from the optimal cost. This factor is called *approximation ratio*. We have the following definitions. We denote the cost of an algorithm ALG on an input I by $\text{ALG}(I)$. The optimal cost of this input is denoted by $\text{OPT}(I)$.

Definition 1.1 *An algorithm $\mathcal{A}$ for a minimization problem Π has an approximation ratio of ρ*

$$\mathcal{A}(I) \leq \rho\text{OPT}(I)$$

for any problem instance I of Π.

The definition for maximization problems is analogous, except that we now require $\text{OPT}(I) \leq \rho\mathcal{A}(I)$ for any input I. Thus the approximation ratio is always a number which is greater than 1. After all, an approximation ratio of 1 is not possible for NP-complete problems, unless P = NP. We call an algorithm that runs in polynomial time and has approximation ratio ρ a ρ-approximation algorithm.

Some problems have the property that for every $\varepsilon > 0$, it is possible to find a solution which has cost within a factor of $1 + \varepsilon$ of the optimal cost.

Definition 1.2 *A family of algorithms which takes as input a problem instance I and a desired accuracy $\varepsilon > 0$, runs in time which is polynomial in the size of the input for any $\varepsilon > 0$ and gives as output a solution which has cost at most $(1 + \varepsilon)\text{OPT}(I)$ is called a* polynomial-time approximation scheme *(PTAS).*

A *fully* polynomial-time approximation scheme is a PTAS which has running time polynomial in both the size of the input I and in $1/\varepsilon$.

1.2 On-line algorithms

In many situations, decisions need to be made without full knowledge of the problem at hand. This holds in particular, when the result of a decision depends on future events. We can model

such situations by online problems. An online problem is characterized by an incrementally appearing input, where the input needs to be processed in the order in which it becomes available and without knowledge of the rest of the input. An online algorithm is simply a list of rules for processing such an input.

There are two ways in which an input can appear incrementally. The input can be given as a list, where the remainder of the list (including its length) is hidden from the online algorithm, or events can occur over time. We will encounter both of these types of inputs in this thesis.

Example A natural and important example of a problem with incomplete information is *paging*, the problem of maintaining a small cache of fast memory in a computer system. In this problem, a controller has to decide which page to eject from the cache when a program requests a page that is not currently in the cache. If future requests are known, this is solved optimally by ejecting the page which will be requested the last among all pages in the cache. However, in real-life applications the sequence of future requests will not be known, and the decision has to be made in some other way. This problem has received a lot of attention over the years [3, 20, 23, 38, 103, 146].

There are many ways of measuring the performance of online algorithms. In this thesis, we will focus on the worst-case behaviour of algorithms. Since the cost for a particular instance may be arbitrarily high, the behaviour of an algorithm should be compared to other algorithms to get meaningful results. In particular, it is important to know how much worse an algorithm performs relative to an *optimal* algorithm, in other words, how much worse it is than the optimal solution for any given problem instance. This kind of analysis is known as *competitive analysis*, which was introduced by Sleator and Tarjan [146]. It involves comparing the performance of an on-line algorithm to the performance of an off-line algorithm that knows the entire problem instance in advance. We do not impose limits on the computational complexity of either the off-line or the on-line algorithm. Therefore the off-line algorithm can always generate an optimal solution.

This type of analysis can be viewed as a game between two players, an on-line algorithm and an *adversary* that both generates the problem instance and serves it as an off-line algorithm. The adversary tries to maximize its performance relative to the on-line algorithm.

Many problems have been studied using competitive analysis. Apart from the paging problem, these include a variety of scheduling problems, bin packing, routing and admission control on a network [8, 22, 83, 3, 71, 7].

The competitive ratio We consider both algorithms that seek to minimize a cost and algorithms that seek to maximize a benefit. We denote the cost or benefit of an algorithm $\mathcal{A}$ on an input sequence σ by $\mathcal{A}(\sigma)$. (The input sequence can appear over time or sequentially.) The optimal off-line cost for an input sequence σ is denoted by $\text{OPT}(\sigma)$.

We compare the ouptut of an on-line algorithm $\mathcal{A}$ to $\text{OPT}(\sigma)$ using the *competitive ratio* [146], which for algorithms that try to minimize a certain cost is defined as follows:

$$\mathcal{R}(\mathcal{A}) = \sup_{\sigma} \frac{\mathcal{A}(\sigma)}{\text{OPT}(\sigma)}$$

where the supremum is taken over all possible inputs. For algorithms that try to maximize a certain benefit, we define the competitive ratio as

$$\mathcal{R}(\mathcal{A}) = \sup_{\sigma} \frac{\text{OPT}(\sigma)}{\mathcal{A}(\sigma)}.$$

In both cases, the best on-line algorithm for a problem is the one that has the lowest possible competitive ratio, and this ratio is at least 1 for any problem.

The competitive ratio of a problem is defined as $\inf_{\mathcal{A}} \mathcal{R}(\mathcal{A})$. The goal is to find an algorithm with competitive ratio close to $\inf_{\mathcal{A}} \mathcal{R}(\mathcal{A})$.

The competitive ratio is clearly a worst-case measure, and by determining the competitive ratio of a certain problem, one can determine the benefit of knowing the entire problem instance in advance. An advantage of such a comparison is that if one can prove that an algorithm has a competitive ratio of $\mathcal{R}$, then any other algorithm can do at most a factor of $\mathcal{R}$ better on any input.

The asymptotic performance ratio In bin packing problems, we are usually interested in the performance of algorithms on "typical" instances, for which the optimal cost increases with the size of the input n. To this end, we now define the *asymptotic performance ratio* (approximation ratio). For a given input sequence σ, let $\text{cost}_{\mathcal{A}}(\sigma)$ be the number of bins used by algorithm $\mathcal{A}$ on σ. Let $\text{cost}(\sigma)$ be the minimum possible number of bins used to pack items in σ. The *asymptotic performance ratio* for an algorithm $\mathcal{A}$ is defined to be

$$\mathcal{R}_{\mathcal{A}}^{\infty} = \limsup_{n\to\infty} \sup_{\sigma} \left\{ \frac{\text{cost}_{\mathcal{A}}(\sigma)}{\text{cost}(\sigma)} \middle| \text{cost}(\sigma) = n \right\}.$$

Note that this ratio can be calculated both for online and for offline problems. It is also known as the *asymptotic worst-case ratio*.

1.3 Multidimensional packing

In the simplest (one-dimensional) version of this problem, we receive a sequence σ of pieces $p_1, p_2, \ldots, p_n$. Each piece has a fixed size in $(0, 1]$. We have an infinite number of bins each with capacity 1. Each piece must be assigned to a bin. Further, the sum of the sizes of the pieces assigned to any bin may not exceed its capacity. The goal is to minimize the number of bins used.

The study of multidimensional packing problems gained an increasing interest in the last few years [17, 28, 47]. A main trend was the study of offline and online packing algorithms for oriented items which are rectangles or boxes. Given a large supply of bins which are squares, or cubes, or a strip of infinite height, the goal is to pack items efficiently, without rotation, such that the sides of all items are aligned with the sides of the strip or the bins.

There are thus two main versions of d-dimensional packing problems:

- **Strip packing** ($d = 2$ or $d = 3$). Here the items need to be packed into a strip of unbounded height (the base is a unit interval or a unit square). Thus each item must be assigned a position such that the item is entirely contained within the strip and does not overlap with any other item. The goal is to minimize the *maximum height used* for any item. The strip packing problem has many applications, for instance cutting objects out of a strip of material in such a way that the amount of material wasted is minimized.

- **Box packing**. We have an infinite number of *bins*, each of which is a d-dimensional unit hyper-cube. Each item $p = (s_1(p), \ldots, s_d(p))$ must be assigned to a bin and a position $(x_1(p), \ldots, x_d(p))$, where $0 \leq x_i(p)$ and $x_i(p) + s_i(p) \leq 1$ for $1 \leq i \leq d$. Further, the positions must be assigned in such a way that no two items in the same bin overlap. A bin is empty if no item is assigned to it, otherwise it is used. The goal is to minimize the *number of bins used*.

We also consider the version where *rotations* are allowed. Although the possibility of allowing rotations was already mentioned by [43], there has been relatively little research into this subject from a worst-case perspective until recently. In the above-mentioned application of strip packing, allowing rotations corresponds to assuming that the material used for cutting is featureless (i.e., the orientation of the items on the strip does not matter). In practice, it is often important that cutting takes place along horizontal or vertical lines. We therefore focus on the case where only $90°$ rotations are allowed.

In the *bounded space* variant of the box packing problem, an algorithm has only a constant number of bins available to accept items at any point during processing. The bounded space assumption is a quite natural one, especially so in online box packing. Essentially the bounded space restriction guarantees that output of packed bins is steady, and that the packer does not accumulate an enormous backlog of bins which are only output at the end of processing.

Known results Offline bin packing has received a great deal of attention, for a survey see [42]. The most prominent results are as follows: Johnson [98] was the first to study the approximation ratios of both online and offline algorithms. Fernandez de la Vega and Lueker [67] presented the first approximation scheme for bin packing. Karmarkar and Karp [104] gave an algorithm which uses at most $\text{cost}(\sigma) + \log^2(\text{cost}(\sigma))$ bins.

The classic (one-dimensional) online bin packing problem was first investigated by Ullman [153]. He showed that the FIRST FIT algorithm has performance ratio $\frac{17}{10}$. This result was then published in [79]. Johnson [99] showed that the NEXT FIT algorithm has performance ratio 2. Yao showed that REVISED FIRST FIT has performance ratio $\frac{5}{3}$, and further showed that no online algorithm has performance ratio less than $\frac{3}{2}$ [159]. Brown and Liang independently improved this lower bound to 1.53635 [25, 123]. The lower bound currently stands at 1.54014, due to van Vliet [156]. Define

$$\pi_{i+1} = \pi_i(\pi_i - 1) + 1, \qquad \pi_1 = 2,$$

and

$$\Pi_\infty = \sum_{i=1}^{\infty} \frac{1}{\pi_i - 1} \approx 1.69103.$$

Lee and Lee presented an algorithm called HARMONIC, which uses $m > 1$ classes and uses bounded space. The fundamental idea of HARMONIC is to first classify items by size, and then pack an item according to its class (as opposed to letting the exact size influence packing decisions).

For the classification of items, we need to partition the interval $(0, 1]$ into subintervals. The standard HARMONIC algorithm uses $M - 1$ subintervals of the form $(1/(i + 1), 1/i]$ for $i = 1, \ldots, M - 1$ and one final subinterval $(0, 1/M]$. Each bin will contain only items from one subinterval (type). Items in subinterval i are packed i per bin for $i = 1, \ldots, M - 1$ and the items in interval M are packed in bins using NEXT FIT(i.e. a greedy algorithm that opens a new active bin whenever an item does not fit into the current active bin, and never uses the previous bins).

For any $\varepsilon > 0$, there is a number M such that the HARMONIC algorithm that uses M classes has a performance ratio of at most $(1 + \varepsilon)\Pi_\infty$ [114]. Lee and Lee also showed there is no bounded space algorithm with a performance ratio below Π_∞.

In Chapter 2, we present a survey on multidimensional packing.

1.4 Scheduling

In the standard scheduling problem, n jobs with different processing requirements are to be scheduled on one machine or on m parallel identical machines. Jobs arrive over time and each job has to be assigned to one of the machines and run there continuously until it is completed. Each machine can only run one job at a time. In the online problem, the online algorithm only becomes aware of a job when it arrives. We also consider the case where the jobs arrive one by one (in a list) and each job arrives only after the previous one has been assigned.

The input is a job sequence $\sigma = \{J_1, \ldots, J_n\}$. Each job J_i arrives at its *release time* r_i and needs to be run for w_i time on one of the machines (w_i is the *size* or *weight* of J_i). J_i is completed after it has been running for w_i time, and the time at which this happens is its *completion time* c_i. The output of an algorithm is a schedule π that for each job determines when and on which machine it is run.

Problem variations We can allow an algorithm to *preempt* a job, halting its execution and continuing it later, possibly on a different machine. We will also consider *related* machines, where each machine has a speed which determines how long it takes to complete a job: on a machine with speed s, a job of size w completes in w/s time. Furthermore, we will consider *variable-speed* machines, where the speed of a machine can be changed at any time, but the required power grows with some power of the speed. The exact relationship between power and speed depends on the device at hand, but for most devices it is of the form s^α for some value $\alpha > 1$. We will assume that there is a predetermined amount of total energy available, and it needs to be allocated to machines and jobs to optimize the schedule. Our results hold for any value of $\alpha > 1$.

In some applications, jobs are not independent of eachother, but instead some jobs may only start after certain other jobs have run. We model this by *precedence constraints*. These can be

represented by a graph, where there is a directed edge between two nodes if the job associated with the second node may only start after the job associated with the first node has finished.

Optimality criteria We will discuss several criteria by which machine scheduling algorithms can be measured. This is first of all the maximum completion time $\max c_i$, the time at which the last job completes. This is also known as the *makespan*. In the case that jobs arrive in a list in stead of over time, the problem of minimizing the maximum makespan is equivalent to minimizing the maximum load over all the machines: *load balancing*. Here a job size does not represent the time that the job is running, but rather the amount that this job adds to the load of a machine when it is assigned to it. We consider the situation where jobs can be split into a limited amount of parts, and give online algorithms for machines of two speeds that are in some cases optimal.

For problems where jobs arrive over time, we also consider the total completion time $\sum c_i$. In particular, we consider the question of how to use *restarts* effectively to minimize $\sum c_i$ in an online environment on a single machine. A restart is weaker than a preemption in that the work done on a job is lost in case of a restart, and the job has to be run again from scratch. Prior to our work, nothing positive was known about this.

Prefetching and caching We also consider a related problem which is called prefetching and caching. This is a classical technique for dealing with memory hierarchies. Prefetching hides access latencies by loading pages into the cache before they are actually required [65, 100, 111, 155]. Caching avoids I/Os by holding pages that are needed again later [20, 23, 69, 135]. Since both techniques compete for the same memory resources, it makes sense to look at the integrated problem [4, 27, 101, 91, 108].

As a concrete (simplified) example consider a flight simulator. Externally stored objects could be topograhical data, textures, etc. At any particular time, a certain set of objects is required in order to play out the right screen content and sound without delays. A demo run could be preplanned resulting in an offline version of the problem. User interactions will result in an online problem where we have a certain lookahead because an action of the user will predetermine the screen content for some small amount of time.

Known results Minimizing the *makespan* for the case that the jobs arrive one by one (load balancing) was considered in a series of papers [83, 84, 19, 102, 3]. Graham [83] introduced the algorithm GREEDY. This algorithm schedules each arriving job on the least loaded machine. The load of a machine is the sum of the loads of the jobs that are assigned to it. Graham showed that GREEDY has a competitive ratio of $2 - 1/m$, which is optimal for $m = 2$ and $m = 3$. Currently, the best upper bound for general m is $1 + \sqrt{(1 + \ln 2)/2} \approx 1.920$ due to Fleischer and Wahl [71] and the best lower bound is 1.853 [82] based on Albers [3].

Chakrabarti, Phillips, Schulz, Shmoys, Stein and Wein [32] gave a 4-competitive algorithm for minimizing the *total completion time* on parallel machines when jobs arrive over time, while Vestjens [154] showed a lower bound of 1.309. For a single machine, Hoogeveen and Vest-

jens [90] gave a 2-competitive on-line algorithm and showed that it is optimal. Two other optimal algorithms were given by Phillips, Stein and Wein [136] and Stougie [151].

Using randomization, it is possible to give an algorithm of competitive ratio $e/(e-1) \approx 1.582$ [33] which is optimal [152]. Vestjens showed a lower bound of 1.112 for deterministic algorithms that can restart jobs [154]. This was improved to 1.2108 by Epstein and van Stee [60].

The offline *splittable jobs* problem was studied by [144]. They showed that the problem is NP-hard (already for identical machines) and gave a PTAS for uniformly related machines. The problem was also studied by [112] who gave an exact algorithm which has polynomial running time for any constant number of uniformly related machines. A different model that is related to our model is scheduling of parallel jobs. In this case, a job has several identical parts that must run simultaneously on a given number of processors [66, 133].

We discuss previous work on prefetching and caching problems in Chapter 9.

1.5 Outline of the thesis

Having discussed the topics of this thesis, we now give an outline of the thesis and its main results.

Part I: Multidimensional packing

- Survey (Chapter 2). In this chapter, we give a survey of the results that have appeared for the several versions of multidimensional bin packing.

- Square packing (Chapter 3). In this problem, all input items are squares, which need to be packed into bins (unit squares) using only orthogonal packings. We present an algorithm for square packing with an absolute worst-case ratio of 2, which is optimal provided P $\neq$ NP.

- Bounded space multidimensional bin packing (Chapter 4). Items are now hyperboxes and need to be packed into multidimensional bins. We present a bounded space algorithm and show that this algorithm is also optimal, with an asymptotic performance ratio of $(\Pi_\infty)^d$. This solves the problem of how to pack hyperboxes using only bounded space, which had been open since 1993. Additionally, we present optimal online bounded space algorithms for several variations of this problem.

- Strip and bin packing with rotations (Chapter 5). Here, it is allowed to *rotate* the items to be packed by $90°$. We give results for six different packing problems with rotations: two-dimensional strip and bin packing, three-dimensional strip and bin packing, and the so-called ”This side up” problem in a three-dimensional strip and in three-dimensional bins.

Part II: Scheduling

- Minimizing the total completion time (Chapter 6). We show how to use restarts on a single online machine to get an algorithm with competitive ratio $3/2$. Without restarts, a ratio better than 2 is not possible, and there are algorithms that have a ratio of 2 [90, 136, 151]. Ours is the first algorithm to break the barrier of 2 and thus the first algorithm that uses restarts efficiently for this goal function.

- Splittable tasks (Chapter 7). We consider jobs that need to be scheduled on m machines and that can be split into at most ℓ parts. On identical machines, we show how to improve on a simple greedy-type algorithm. For the case where a subset of the machines has speed $s > 1$, we give an algorithm which is optimal for sufficiently large ℓ.

- Speed scaling (Chapter 8). We give an $O(\log m)$-approximation algorithm for minimizing the makespan for job with precedence constraints on m parallel variable-speed machines, where there is a global bound on the amount of energy available.

- Prefetching and caching (Chapter 9). We present a new theoretical model for this problem. For this model, we present an I/O-optimal algorithm which uses a “semi”-greedy approach and runs in quadratic time. Additionally we consider the online problem. We show that competitive algorithms are possible using resource augmentation on the speed *and* lookahead, and we provide a tight relationship between the amount of resource augmentation on the speed and the amount of lookahead required.

An overview of the most important notations is given in Table 1.5.

1.6 Credits

In this section we list the papers on which the chapters are based.

Chapter 2 is based on Leah Epstein and Rob van Stee, Multidimensional packing problems, to appear in Teofilo Gonzalez (Editor), *Approximation Algorithms and Metaheuristics*.

Chapter 3 is based on Rob van Stee, An approximation algorithm for square packing, *Operations Research Letters*, 32(6):535–539, 2004.

Chapter 4 is based on Leah Epstein and Rob van Stee, Optimal online algorithms for multidimensional packing problems, *SIAM Journal on Computing*, 35(2):431–448, 2005.

Chapter 5 is based on Klaus Jansen and Rob van Stee, On strip packing with rotations, in *Proc. of 37th ACM Symposium on Theory of Computing (STOC 2005)*, p. 755–761, ACM, 2005, and on Leah Epstein and Rob van Stee, This side up! *ACM Transactions on Algorithms*, 2(2):228-243, 2006.

Chapter 6 is based on Rob van Stee and Johannes A. La Poutré, Minimizing the total completion time on a single on-line machine, using restarts, *Journal of Algorithms*, 57(2):95–129, 2005.

Chapter 7 is based on Leah Epstein and Rob van Stee, Online scheduling of splittable tasks, *ACM Transactions on Algorithms*, 2(1):79–94, 2006.

$\mathcal{A}$	an (approximation or online) algorithm
σ	input sequence for the algorithm, e. g. a job sequence
$\mathcal{A}(\sigma)$	cost or benefit of algorithm $\mathcal{A}$ on input σ
OPT	optimal (off-line) algorithm
$\mathcal{R}(\mathcal{A})$	competitive ratio of $\mathcal{A}$
n	number of items in the input σ
p_i	the ith item to be packed (square, box, hyperbox)
$s_j(p)$	size of item p in the jth dimension
$v(p)$	volume (or area) of item p
$x_j(p)$	position of item p in the jth dimension (in a certain packing)
$w(p)$	weight of item p
$w_j(p)$	weight of item p in the jth dimension (where applicable)
$t(p)$	type of item p
$t_j(p)$	type of item p in the jth dimension
m	number of machines (or number of off-line machines)
J_i	the ith job
r_i	its release time
w_i	its size or weight
c_i	its completion time
s_i	the speed at which it runs (in Chapter 8)
p_i	the power at which it runs (in Chapter 8)

Table 1.1: An overview of the notation. The top section defines notation for the entire thesis, the middle section is for Part I (multidimensional packing) and the third section is for part II (scheduling).

Chapter 8 is based on Kirk Pruhs, Rob van Stee and Patchrawat Uthaisombut, Speed scaling of tasks with precedence constraints, in *Proc. 3rd Workshop on Approximation and Online Algorithms (WAOA 2005)*, p. 307–319, volume 3879 of *Lecture Notes in Computer Science*, Springer, 2006. To appear in *Theory of Computing Systems*.

Chapter 9 is based on Peter Sanders, Johannse Singler, and Rob van Stee, Real-time prefetching and caching, manuscript.

Part I

Multidimensional packing

Chapter 2

Multidimensional packing problems: a survey

As stated in the Introduction, there are several ways to generalize the bin packing problem to more dimensions. In this chapter, we consider two- and three-dimensional strip packing, and bin packing in dimensions two and higher. Finally we consider vector packing and several other variations.

In the most common two-dimensional version, the items are rectangles or squares, and the bins are unit squares. In the strip packing probem, instead of bins, we are given a strip of width 1 and unbounded height. In higher dimensions, the rectangles are replaced by boxes (or hyperboxes), the squares by cubes (or hypercubes), and the unit square by a unit cube (or hypercube of the relevant dimension). Strip packing becomes column packing.

A striking difference between one-dimensional bin packing and its multidimensional generalizations is that while for one-dimensional bin packing, offline algorithms clearly outperform online algorithms, this is not always the case in more dimensions. There are several cases where an online algorithm was at one point the best known approximation algorithm, or remains the best known approximation until today. Most likely, this simply reflects the fact that we do not understand the multidimensional case as well as the one-dimensional case. One the other hand, some results simply cannot be generalized. For instance, we now know that there cannot be an APTAS for two-dimensional bin packing [17], or for two-dimensional vector packing [158].

An important special case in multidimensional bin and strip packing is the case where (hyper-)cubes need to be packed. For this case, better results are known than for the general case. In particular, the offline version of this problem admits an APTAS [17, 47].

As is the case for one-dimensional bin packing, most attention has gone to the asymptotic worst-case ratio, but in the course of this chapter we will encounter some results on the absolute ratio as well.

Rotations When packing of rectangle or boxes is considered, there are several ways to define the problem. In the oriented problem, items have a fixed orientation, and cannot be rotated. In the rotatable (or non oriented) version, an item can be rotated and placed in any position such that its sides are parallel to the sides of the bin. Finally there are mixed versions where items

can be rotated in certain directions, but not all directions. One such three-dimensional model where items can be rotated to the left or to the right but the top and bottom must remain such is the "z-oriented" packing [129, 131] studied by Miyazawa and Wakabayashi, also known as the "This Side Up" problem.

An illustration of the difference between the two problems is given in Figure 2.1. In this figure we see packings of rectangles of sides $\frac{3}{5}$ and $\frac{2}{5}$. If the rectangles are oriented so that their height is $\frac{3}{5}$ and cannot be rotated, we can pack at most two such items in one bin. However, if rotation is allowed, we can pack as much as four such rectangles together in one bin.

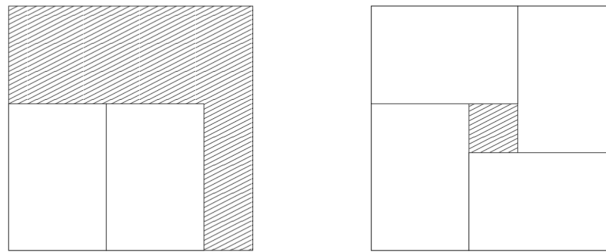


Figure 2.1: A comparison between the oriented and the rotatable models

This chapter is organized as follows. We begin by presenting the algorithm Next Fit Decreasing Height (NFDH), which is a fundamental algorithm for two-dimensional packing problems, in Section 2.1. We then discuss results on multidimensional packing problems, in order of increasing dimension. That is, we start with strip packing in Section 2.2 and move to two-dimensional bin packing in Section 2.3. We then discuss column packing in Section 2.4 and three- and more-dimensional bin packing in Section 2.5. Finally, we mention results on vector packing in Section 2.6 and discuss several variations on multidimensional packing in Section 2.7.

2.1 Next Fit Decreasing Height

In 1968, Meir and Moser [126] introduced an algorithm for packing d-dimensional cubes into a d-dimensional hyperbox, which they called Next Fit Decreasing (NFD). This algorithm sorts the cubes by decreasing volume and packs them into layers. The authors show that if the sides of the cubes are denoted by $x_1, x_2, \ldots$, and they are packed into a hyperbox of sides $a_1, \ldots, a_d$ where $x_1 \leq a_i$ for $i = 1, \ldots, d$, then the cubes can be packed into the hyperbox as long as their total volume is at most

$$x_1^d + \prod_{i=1}^{d}(a_i - x_1).$$

For $d = 2$ (packing squares into a rectangle), the algorithm works as follows. The largest square is put in the bottom left corner of the rectangle. The height of the first layer is equal to the side of this square. The next squares are put in this layer, next to each other and touching each other and the bottom of the layer, until one does not fit. At this point we define a new layer above the first layer, with height equal to the side of the first square packed into it. This

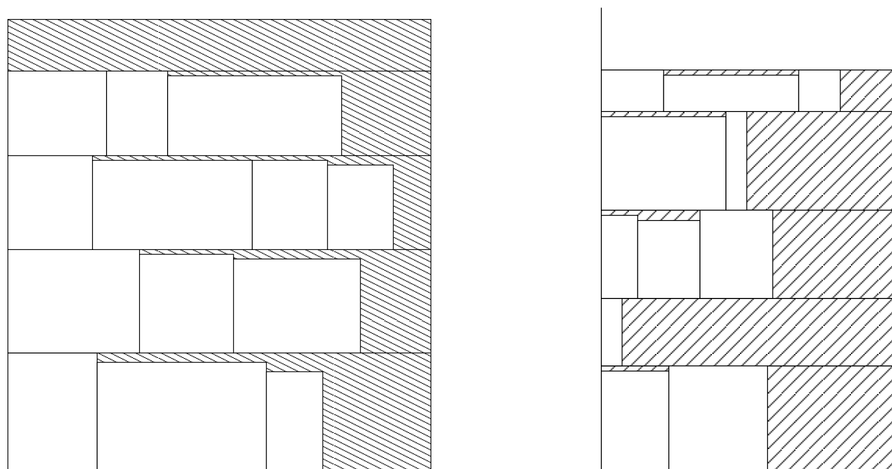


Figure 2.2: An illustration of a packing of NFDH (left) and of a shelf packing algorithm (right).

continues until all squares are packed, or there is not enough room to pack some item (it does not fit into the current layer, and the last layer that is left is either empty or not high enough).

This algorithm (for two dimensions) was extended to an algorithm for packing rectangles into a rectangle (or a strip) by Coffman, Garey, Johnson and Tarjan [43], which was called Next Fit Decreasing Height (NFDH). It sorts the rectangles by decreasing height and then packs them as above.

They showed that if this algorithm is applied to pack rectangles into a strip (of unbounded height), then the height used to pack the rectangles is at most twice the optimal height, plus an additive constant which is equal to the height of the highest rectangle. (Thus its absolute worst-case ratio is 3.)

The proof is quite straightforward. In each level, there may be wasted space to the right of the rightmost item, and above all items except the first. The height of a level is the height of the first item in it. This item did not fit on the previous level. This implies that the total area of the items in level i plus the first item in level $i+1$ is at least the height of level $i+1$ (since the width of the strip is 1). (If we move all items in level i up to level $i+1$, and shift the first item in level $i+1$ to the right, then level $i+1$ is entirely covered by items.)

Adding up the heights of all levels, this is upper bounded by twice the area of the packed items plus the height of the first level. This explains the performance bound including the additive constant, since the total area is an obvious lower bound for the optimal height.

This fundamental algorithm was used in many later papers as a subroutine. It works especially well when all rectangles are guaranteed to have a small width (relative to the width of the strip), and this property was for instance used by Kenyon and Rémila [107] in their approximation scheme for strip packing.

Meir and Moser also showed the following important result in the same paper [126].

Theorem 2.1 *Any set of rectangles with sides at most x and total area A can be packed into any rectangle of size $a \times b$ if $a \geq x$ and $ab \geq 2A + a^2/8$. This result is best possible.*

For packing rectangles into a unit square, this result states that any set of rectangles of total area at most $7/16$ (and sides not larger than 1) can be packed into a unit square.

2.2 Strip packing

2.2.1 Online results

Baker and Schwarz [13] were the first to study two-dimensional online strip packing. They introduced a class of algorithms called *shelf algorithms*. A shelf algorithm uses a one-dimensional bin packing algorithm A and a parameter $\alpha \in (0, 1)$. Items are classified by height: an item is in class s if its height is in the interval $(\alpha^{s-1}, \alpha^s]$. Each class is packed in separate *shelves*, where we use A to fill a shelf and open a new shelf when necessary. Note that the algorithm A is not necessarily on-line. See Figure 2.2 for an illustration of a shelf algorithm.

Baker and Schwarz showed that the algorithm FIRST FIT SHELF, which uses FIRST FIT as a subroutine, has an asymptotic performance ratio arbitrarily close to 1.7. Csirik and Woeginger [52] showed that by using HARMONIC as a subroutine, it is possible to achieve an asymptotic performance ratio arbitrarily close to $h_\infty \approx 1.69103$. Moreover, they show that any shelf algorithm, online or offline, has a performance ratio of at least h_∞. The idea of the lower bound is that items are given that could be combined nicely next to each other, but which end up in different height classes and are therefore packed in separate shelves. So basically, the best thing one can do is to use a bounded space algorithm (which has a constant number of simultaneously active bins) like HARMONIC as the subroutine. Finally they mention that from the one-dimensional lower bound of van Vliet [156] together with the insights of Baker, Brown and Katseff [10] implies a general lower bound for online algorithms of 1.5401. It remains an open problem how to improve the upper bound of Csirik and Woeginger. It does not seem easy to find a good on-line algorithm that does not use shelves. As for the absolute performance ratio, Brown, Baker and Katseff [26] showed a lower bound of 2 for any algorithm. They also show some lower bounds for algorithms that may sort they items.

2.2.2 Offline results

The strip packing problem was introduced in 1980 by Baker, Coffman and Rivest [12]. They developed the first offline approximation algorithms for this problem, and give an upper bound of 3 on the absolute performance ratio. This bound was later improved to 2 independently by Schiermeyer [138] and by Steinberg [150], using different approaches. In the same issue of SIAM Journal on Computing, Coffman, Garey, Johnson and Tarjan [43] showed that NFDH has an asymptotic performance ratio of 2, FFDH achieves a value of 1.7, and an algorithm called Split-Fit has $3/2$. Also in 1980, Sleator [147] gave an algorithm with asymptotic performance ratio 2.5, but absolute performance ratio of 2, which is better than that of Split-Fit, which has 3. In 1981, Baker, Brown and Katseff [10] gave an offline algorithm with asymptotic worst case ratio $5/4$. Finally, Kenyon and Rémila [107] designed an asymptotic fully polynomial time approximation scheme.

This scheme uses some nice ideas, which we describe below.

Fractional strip packing A fractional strip packing of L is a packing of any list L' obtained from L by subdividing some of its rectangles by horizontal cuts: each rectangle (w_i, h_i) is replaced by a sequence of rectangles $(w_i, h_i^1), (w_i, h_i^2), \ldots, (w_i, h_i^{k_i})$ such that $\sum_{j=1}^{k_i} h_i^j = h_i$.

In the case that L contains only items of m distinct widths in $(\varepsilon', 1]$, where $\varepsilon' > 0$ is some constant, it is possible to find a fractional strip packing of L which is within 1 of the optimal fractional strip packing $\text{FSP}(L)$ in polynomial time. Moreover, it is possible to turn this packing into a regular strip packing at the loss of only an additive constant $2m$. Denote the height of the optimal strip packing for L by $\text{OPT}(L)$. We conclude that we find a packing with height at most

$$\text{FSP}(L) + 1 + 2m \le \text{OPT}(L) + 2m + 1 \tag{2.1}$$

Modified NFDH (Next Fit Decreasing Height) This is a method for adding narrow items (items of width at most ε') to a packing of wide items such as described above. Such a packing leaves empty rectangles at the right hand side of the strip. Each of these rectangles is packed with narrow items using NFDH (starting with the highest narrow item in the first rectangle). When all rectangles have been used, the remaining items (if any) are packed above the packing using NFDH on the entire width of the strip.

First Fit Decreasing Height (FFDH) FFDH is a natural variation on NFDH, which each time uses First Fit to find a level for the current item to be packed. The following theorem was proved by Coffman et al. [43].

Theorem 2.2 *Let L be any list of rectangles ordered by non-increasing height such that no rectangle in L has width exceeding $1/m$ for some $m \ge 2$. Then*

$$FFDH(L) \le (1 + 1/m)A(L) + 1,$$

where $A(L)$ is the total area of the items in L.

Grouping and rounding This method is a variation on the linear rounding defined by Fernandez de la Vega and Lueker [67]. It works as follows.

We stack up the rectangles of L by order of non-increasing widths to obtain a left-justified stack of total height $h(L)$. We define $m-1$ threshold rectangles, where a rectangle is a threshold rectangle if its interior or lower boundary intersects some line $y = ih(L)/m$ for some $i \in \{1, \ldots, m-1\}$. We cut these threshold rectangles along the lines $y = ih(L)/m$. This creates m groups of items that have height exactly $h(L)/m$.

First, the widths of the rectangles in the first group are rounded up to 1, and the widths of the rectangles in each subsequent group are rounded up to the widest width in that group. This defines L_+.

Second, the widths of the rectangles in each group are rounded down to the widest width of the next group (down to 0 for the last group). This defines L_-.

It is easy to find a strip packing for L_- using a reduction to fractional strip packing. Moreover, it can be seen that the stack associated with L_+ is exactly the union of a bottom part of

width 1 and height $h(L)/m$ and the stack associated with L_-. Thus

$$\text{FSP}(L) \leq \text{FSP}(L_+) = \text{FSP}(L_-) + h(L)/m. \tag{2.2}$$

Partial ordering We say that $L \leq L'$ if the stack associated to L (used for the grouping above), viewed as a region of the plane, is contained in the stack associated to L'. Note that $L \leq L'$ implies that $\text{FSP}(L) \leq \text{FSP}(L')$. As an example, in the grouping above we have $L_- \leq L \leq L_+$.

2.2.3 Rotations

The upper bound of 2 of NFDH and Bottom Leftmost Decreasing Width (BLDW) remain valid if orthogonal rotations are allowed, since the proofs use only area arguments. Miyazawa and Wakabayashi [131] presented an algorithm with asymptotic approximation ratio of 1.613. In Chapter 5, Section 5.1.1, we present a simpler algorithm which achieves an asymptotic approximation ratio of $3/2$. This algorithm packs items that are wider and higher than $1/2$ optimally, and packs remaining items first next to this packing (where possible) and finally on top of this packing. In this way, the resulting packing is either optimal, or almost all heights a width of $2/3$ is occupied by items. Finally, approximation schemes were given by Jansen and van Stee [94]. We present the combinatorial polynomial-time approximation scheme from this paper in Chapter 5, Section 5.1.2.

2.3 Two-dimensional bin packing

We saw in section 2.2.1 that we can use a one-dimensional bin packing algorithm as a subroutine for a strip packing algorithm, basically without a loss in (asymptotic) performance ratio. Similarly, a two-dimensional bin packing algorithm can be used as a subroutine to create a three-dimensional strip packing algorithm, and this also holds for higher dimensions.

On the other hand, a d-dimensional strip packing algorithm can also be used to create a d-dimensional bin packing algorithm at a cost of a factor of two in the performance ratio. The idea is to cut the packing generated by the strip packing algorithm into pieces of unit height. For each piece we do the following. Items that are completely contained in the piece are put together in one bin. Items that are partially in the next piece are put together in a second bin. See Figure 2.3.

Say we have a guarantee of R on the asymptotic performance ratio of the strip packing algorithm. Then this method gives us $2R \cdot \text{OPT}(L) + C$ bins for an input L, where $\text{OPT}(L)$ is the height of the optimal strip packing. On the other hand, there cannot be a *bin* packing into less than $\text{OPT}(L)$ bins, because this packing could be trivially turned into a strip packing of height less than $\text{OPT}(L)$. This explains the factor of two loss.

2.3.1 Online results

Coppersmith and Raghavan were the first to study the online version of this problem. They gave an online algorithm with asymptotic performance ratio of 3.25 for $d = 2$ (and 6.25 for

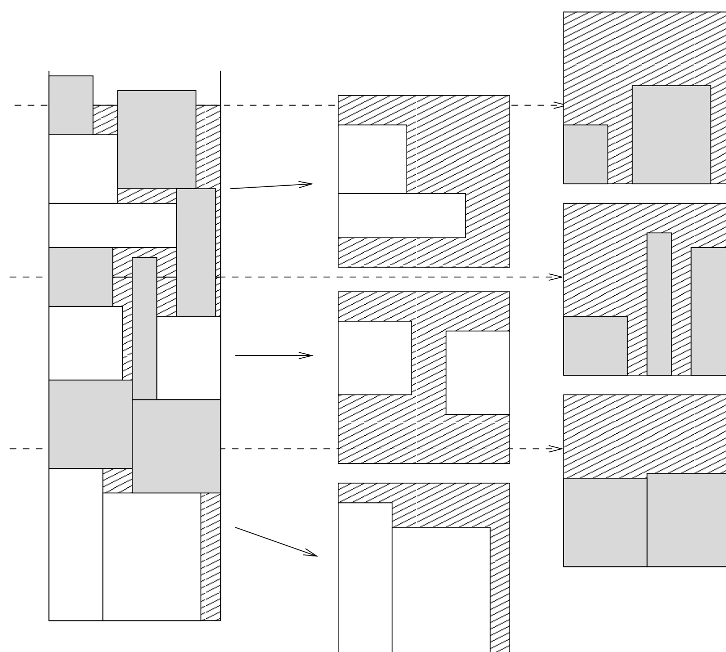


Figure 2.3: Converting a packing in a strip into a packing in bins

$d = 3$) [46]. This result was improved by Csirik et al., who presented an algorithm with performance ratio 3.0625 [50]. In the same year, Csirik and van Vliet showed an online bin packing algorithm for arbitrary dimensions, which achieves a performance ratio of h_∞^d, where d is the dimension [51]. Note that already for $d = 2$, this improves over the previous result, since $h_\infty^2 \approx 2.85958$. (See also [118] for $d = 2, 3$.) Finally, Seiden and van Stee [141] gave an algorithm with ratio 2.66013 for two-dimensional bin packing.

In Chapter 4, Section 4.2, we describe a new technique for packing small multidimensional items online, enabling us to achieve the asymptotic performance ratio of h_∞^d [51] using only bounded space.

Galambos [75] was the first to give a lower bound for this problem which was higher than the best known lower bound for one-dimensional bin packing. His bound was 1.6. This was later successively improved to 1.808 by Galambos and van Vliet [77], 1.851 by van Vliet [157], and finally to 1.907 by Blitz, van Vliet, Woeginger [21]. The gap between the upper and lower bounds remains relatively large to this day, and it is unclear how to improve either of them significantly.

An interesting special case is where all items are squares. Coppersmith and Raghavan [46] showed that their algorithm has an asymptotic performance ratio of 2.6875 for this case, and gave a lower bound of $4/3$. This lower bound actually holds for the more general problem of packing hypercubes. Seiden and van Stee [141] showed that the algorithm HARMONIC $\times$ HARMONIC, which uses the HARMONIC algorithm to find slices for items, and then uses the HARMONIC algorithm again to find bins for slices, has an asymptotic performance ratio of at most 2.43828. They gave a lower bound of 1.62176 for any online algorithm, and also showed a lower bound of 2.28229 for bounded space algorithms using the same instances.

Epstein and van Stee [62] give an algorithm with asymptotic performance ratio at most 2.24437, and improved the lower bound to 1.6406. The upper bound was recently improved to 2.1439 by Han, Ye and Zhou [88]. Here too, the gap between the lower and the upper bounds remains disappointingly large. Finally, Epstein and van Stee [63] give bounds for the performance of the optimal bounded space algorithm from Chapter 4, Section 4.1, showing that its performance ratio lies between 2.3638 and 2.3692.

2.3.2 Offline results

As mentioned at the start of this chapter, Bansal and Sviridenko proved that the two-dimensional bin packing problem is APX-hard [17]. Thus, there cannot be an asymptotic polynomial time approximation scheme for this problem.

Chung, Garey and Johnson [40] were the first to give an approximation algorithm for this problem. It has an asymptotic approximation ratio of 2.125. As mentioned above, the APTAS for strip packing by Kenyon and Rémila implies a $(2+\varepsilon)$-approximation for any $\varepsilon > 0$. In 2002, Caprara [28] gave a h_∞-approximation.

Leung et al. [115] proved that the special case of packing squares into squares is still NP-hard (for general two-dimensional bin packing, this follows immediately from the one-dimensional case). Ferreira, Miyazawa, and Wakabayashi [68] gave a 1.988-approximation for this problem, which uses as a subroutine an optimal algorithm for packing items with sides larger than $1/3$. They conjecture that packing items with sides larger than $1/4$ is already NP-hard. Independently of eachother, Kohayakawa et al. [109] and Seiden and van Stee [141] gave a $(14/9+\varepsilon)$-approximation $(1.5555\ldots+\varepsilon)$. However, the first result is more general in that it actually gives a $(2-(2/3)^d+\varepsilon)$-approximation for packing d-dimensional hypercubes. The idea of both these algorithms is to find an optimal packing for large items (items with sides larger than ε) and to add the small items to this packing. Specifically, any bins in the optimal packing which contain only a single item with sides larger than $1/2$ are filled with small items using the algorithm Next Fit Decreasing (NFD) from Meir and Moser (see Section 2.1). It is shown that all other bins are already "reasonably full", leading to the approximation guarantee.

In the same year, Caprara [28] gave an algorithm with performance ratio in the interval $(1.490, 1.507)$ provided a certain conjecture holds. Two years later, Epstein and van Stee [61] gave a $(16/11+\varepsilon)$-approximation $(1.4545\ldots+\varepsilon)$. Simultaneously and independently of each other, Bansal and Sviridenko [17] and Correa and Kenyon [47] presented an asymptotic polynomial time approximation scheme for this problem, which also works for the more general problem of packing hypercubes.

Recently, Bansal, Lodi and Sviridenko [15] showed another special case of the two-dimensional bin packing problem which admits an APTAS. This is rectangle packing, where the packing of each bin must be possible to achieve using guillotine cuts only. That is a sequence of edge to edge cuts, parallel to the edges of the bin. Even more special cases, where the number of stages in the sequence of guillotine cuts is limited, were studied by Caprara, Lodi and Monaci [30]. They designed an APTAS for the two stage problem. Note the shelf packing described above actually uses two stages of guillotine cuts. Kenyon and Rémila [107] point out that their approximation scheme uses five stages of guillotine cuts.

As regards the absolute performance ratio, Zhang [162] gave an approximation algorithm with absolute worst-case ratio of 3 for two-dimensional bin packing. In Chapter 3, we present an absolute 2-approximation for square packing, which is optimal by the result of Leung et al. [115].

2.3.3 Resource augmentation

Since there cannot be an approximation scheme for general two-dimensional bin packing, several authors have looked at the possibility of resource augmentation, i.e., giving the approximation algorithm slightly larger bins than the offline algorithm that it is compared to. Correa and Kenyon [47] give a dual polynomial time approximation scheme. That is, they give a polynomial time algorithm to pack rectangles into the k bins of size $1 + \varepsilon$, where these rectangles cannot be packed in less than k bins of size 1. Bansal and Sviridenko [18] showed that it is possible to achieve this even if the size of the bin is relaxed in one dimension only.

2.3.4 Rotations

For the case where rotations are allowed, Epstein [58] showed an online algorithm with asymptotic performance ratio of 2.45. The online problem was studied before by Fujita and Hada [74]. They presented two online algorithms and claimed asymptotic performance ratios of at most 2.6112 and 2.56411. Epstein [58] mentioned that the proof in [74] only shows that the first algorithm has an asymptotic performance ratio of at most 2.63889 and that the proof of the second algorithm is incomplete.

Two years later, Miyazawa and Wakabayashi [131] gave an offline algorithm with asymptotic performance ratio 2.64. In Chapter 5 (Section 5.2), we present an approximation algorithm with asymptotic performance ratio 2.25. It divides the items into types and combines them into bins such that in almost all bins, an area of $4/9$ is occupied. Correa [48] adapted the dual polynomial time approximation scheme from [47] to rotatable items.

2.4 Column (three-dimensional strip) packing

2.4.1 Online and offline results

Li and Cheng were the first to consider this problem. In their paper [119] from 1990, they showed that three-dimensional versions of NFDH and FFDH have unbounded worst-case ratio. They gave several approximation algorithms, the best of which has an asymptotic performance ratio of 3.25. Their first algorithm sorts the items by height and then divides them into groups of area (in the first two dimensions) at most $7/16$, so that they can be packed into a single layer by Theorem 2.1. They improve on this by classifying items with similar bottoms, and packing similar items together into layers. Two items have similar bottoms if both their length and their width fall into the same class when classified by the HARMONIC algorithm. For the case where all items have square bottoms, the ratio improves to 2.6875.

Two years later, the same authors [117] presented an online algorithm with asymptotic performance ratio arbitrarily close to $h_\infty^2 \approx 2.89$ for three-dimensional strip packing. At the time, there was no better *offline* approximation known. This algorithm uses the HARMONIC algorithm as a subroutine in both horizontal dimensions (i.e. to find a strip for a two-dimensional item, and a place inside a strip for a one-dimensional item), and a geometric rounding for the heights. The paper actually discusses several online algorithms for this problem and only mentions the use of HARMONIC in the summary section. The authors note that the improvement in the asymptotic performance ratio compared to the approximation algorithm from their earlier paper [119] only comes at the cost of a high additive constant.

In 1997, Miyazawa and Wakabayashi [128] improved the offline upper bound to 2.66994 (2.36 for items with square bottoms). This algorithm places columns of similar items next to eachother in the strip, thus avoiding the layer structure of the previous algorithms. The algorithm is quite involved and its description takes three pages. This remains the best result to date.

2.4.2 Rotations

In the case where rotations are allowed, it becomes relevant what exactly the dimensions of the strip are. In two-dimensional strip packing, this does not really play a part, but in column packing, the base of the column might not be a square.

However, if the base is not a square but may be an arbitrary rectangle, then having the ability to rotate items horizontally (leaving the top side unchanged) does not help, as was shown by Miyazawa and Wakabayashi [129]. The idea is that in this case it is possible to scale the input so that the smallest width of an item is still larger than the length of the base of the strip, so that no item can be rotated and still fit inside the strip. For this reason, in this section we focus on the case where the base of the strip is a square.

In Chapter 5 (Section 5.4.1), we give an approximation algorithm with asymptotic worst-case ratio of $9/4 = 2.25$, improving on the upper bound of 2.76 by Miyazawa and Wakabayashi [131]. The special case where only rotations that leave the top side of items at the top are allowed has received more attention. It was introduced by Li and Cheng [116] as a model for a job scheduling problem in partitionable mesh connected systems. Here each job i is given by a triple (x_i, y_i, t_i), meaning that job i needs a submesh of dimensions $x_i \times y_i$ or $y_i \times x_i$ for t_i time units. They give an algorithm for minimizing the makespan (i.e., the height of the packing) which has asymptotic performance bound $4\frac{4}{7}$. This was improved to 2.543 by Miyazawa and Wakabayashi [131]. In Chapter 5 (Section 5.3), we present a 2.25-approximation.

2.5 Three- and more-dimensional bin packing

At present, the online bounded space algorithm from Chapter 4, Section 4.2 is the best (online or offline) algorithm for packing multidimensional items into bins for any dimension $d \geq 3$. Clearly, this problem is APX-hard as well since it includes the two-dimensional bin packing problem as a special case [17].

Blitz, van Vliet, and Woeginger [21] gave a lower bound of 2.111 for online algorithms for

$d = 3$. However, there is no good lower bound known for larger dimensions: nothing above 3. It appears likely that the asymptotic performance bound of any online algorithm must grow with the dimension.

For the special case of packing hypercubes online in dimensions $d \geq 4$, there is no better lower bound than the $4/3$ given by Coppersmith and Raghavan [46] (which works in any dimension $d \geq 2$).

The bounded space algorithm from Chapter 4, Section 4.1 for this problem has a performance ratio which is sublinear in d: it is $O(d/\log d)$ and $\Omega(\log d)$.

For $d = 3$ (online cube packing), Miyazawa and Wakabayashi [130] showed that the algorithm of Coppersmith and Raghavan [46] has an asymptotic performance bound of 3.954. Epstein and van Stee [62] give an algorithm with asymptotic performance ratio at most 2.9421, and a lower bound of 1.6680. The upper bound was improved to 2.6852 by Han, Ye and Zhou [88]. Furthermore, Epstein and van Stee [63] give bounds for the performance of the bounded space algorithm from Chapter 4, Section 4.1, showing that its performance ratio lies between 2.95642 and 3.0672.

As was seen in section 2.3.2, we can do even better offline. Before Bansal and Sviridenko [17] and Correa and Kenyon [47] gave their asymptotic polynomial time approximation scheme for any dimension $d \geq 2$, Miyazawa and Wakabayashi [130] gave two approximation algorithms, of which the best had an asymptotic performance ratio of 2.6681. Soon afterwards, Kohayakawa et al. [109] presented their paper which we discussed in section 2.3.2 as well. For $d = 3$, its asymptotic performance bound is $46/27 + \varepsilon \approx 1.7037\ldots + \varepsilon$.

2.6 Vector packing

In this section we discuss the non-geometric version of multidimensional bin packing. The d-dimensional "vector packing", or "vector bin packing" problem is defined as follows. The bins are instances of the "all-1" vector $(1, 1, \ldots, 1)$ of length d. Items are d-dimensional vectors, whose components are all in $[0, 1]$. A packing is valid if the vector sum of all items assigned to one bin cannot exceed the capacity of the bin (i.e., 1) in any component. Since all bins are identical, the goal is to minimize the number of bins used.

The problem can be seen as a scheduling problem with limited resources. The machines (with correspond to bins) have fixed capacities of several resources as memory, running time, access to other computers etc. The items in this case are jobs that need to be run, each job requires a certain amount of each resource. Another application arises from viewing the problem as a storage allocation problem. Each bin has several qualities as volume, weight etc. Each item requires a certain amount of each quality. Both applications are relevant to both offline and online environments.

For many years there were very few results on this problem. In the first paper which obtained an APTAS for classical bin packing [67], Fernandez de la Vega and Lueker implies a $(d + \varepsilon)$-approximation for the vector packing problem. This improved very slightly on some online results. These results were an upper bound of $d + 1$ on the performance ratio of any algorithm for which the output never has two bins that can be combined, given by Kou and Markowsky

[110], and a tight bound on the performance of First Fit of $d + \frac{7}{10}$, given by Garey et al. [78]. Note that this is a generalization of the tight bound of $\frac{17}{10}$ for First Fit in one dimension.

Since these results were obtained, for a while there was hope that an APTAS would be found for this problem. However, Woeginger proved in [158] that unless $P = NP$, there cannot be such an APTAS, already for two-dimensional vectors. Clearly, more restricted classes of vectors may still admit an APTAS. One such type of input is one where there is a total order on all vectors. In [29], Caprara, Kellerer and Pferschy showed that an APTAS for this problem indeed exists.

The offline result for the general case was finally improved by Chekuri and Khanna [35]. They designed an algorithm of asymptotic performance $1 + \varepsilon d + O(\ln \frac{1}{\varepsilon})$. If d is seen as a constant, the best ratio achieved in this way is $O(\ln d)$. They proved that for an arbitrary d, it is APX-hard to approximate the problem within a factor of $d^{\frac{1}{2}-\varepsilon}$ for every fixed positive ε. This was shown using a reduction from graph coloring.

The online result was not improved since 1976. Lower bounds on the performance ratio of online algorithms, that tend to 2 as d grows, were shown by Galambos, Kellerer and Woeginger [76]. Improved lower bounds were given by Blitz, van Vliet, and Woeginger [21], but this construction also tends to 2 as d grows.

As for the absolute approximation ratio, Kellerer and Kotov [105] designed an algorithm for two-dimensional vector packing with absolute approximation ratio of at most 2. Recently, Erlebach [64] showed a non-constant lower bound for on the absolute performance ratio for this problem. Interestingly, the method is similar to the one used by Chekuri and Khanna to show the hardness of approximation. The lower bound holds for the asymptotic performance ratio if d is not seen as a constant, i.e., for arbitrary d.

As for variable sized packing, the online problem was studied by Epstein [57]. In this problem, the algorithm may use bins out of a given finite subset. This subset contains the standard "all-1" vector, and possibly other vectors. The cost of a bin is the sum of its components. She showed that there exists a finite set where an online algorithm can achieve performance ratio $1+\varepsilon$ (by defining the class of bins to be dense enough), whereas for another set (which contains except for the "all-1" bin only bins that have relatively small components), the ratio must be linear. Clearly, no matter what the set is, there exists a simple algorithm with linear performance ratio.

Analogously to the bin covering problem, we can define the vector covering problem, where the vector sum of all vectors assigned to one bin is *at least* 1 in every component. This problem was studied by Alon et al. [5]. In this paper it was shown that the performance ratio of any online algorithm is at least $d + \frac{1}{2}$. A linear upper bound of $2d$ is achieved by an algorithm which partitions the input into classes. The same paper contains offline results as well. An algorithm of performance guarantee $O(\log d)$ is presented as well as a simple and fast 2-approximation for $d = 2$.

In [56] some results on variable sized vector covering are given. These results focus on cases where all bins are vectors of zeros and ones. The benefit of a covered bin is the sum of its non-zero components. The considered cases for the bins set are as follows. A set which consists of a single type of bin, a set of all unit vectors (all components are zero except for one), unit prefix vectors (some prefix of the vector consists of ones only) and the set of all zero-one vectors.

2.7 Variations

2.7.1 Rectangle stretching

Imreh [92] studied an oriented online strip packing problem where rectangles can be stretched in a way that results in a larger height but the original area. Note that allowing stretching that increases the width makes the problem trivial as all items would be stretched to have the same width as the bin. He showed that the offline problem is polynomially solvable, and that if the online problem is considered under the asymptotic performance ratio measure (and assuming an upper bound of 1 on the original height of any rectangle), then the performance ratio can be made arbitrarily close to 1. Therefore, the main results are for the absolute performance ratio. There are algorithms of performance ratios 6 and 4, and a lower bound of 1.73 on the performance ratio of any online algorithm.

2.7.2 Items appear from the top

A “Tetris like” online model was studied in a few papers. This is similar to strip packing, however, in this model, a rectangle cannot be placed directly in its designated area, but it arrives from the top as in the Tetris game, and should be moved continuously around only in the free space until it reaches its place, (see figure 2.4), and then cannot be moved again.

In [9], the model was introduced by Azar and Epstein. In that paper, both the rotatable and the oriented models were studied. For the rotatable model, a 4-approximation algorithm was designed. The situation for the oriented problem is more difficult, as no algorithm with constant approximation ratio exists for unrestricted inputs. If the width of all items is bounded below by ϵ and/or bounded above by $1-\epsilon$, the authors showed a lower bound of $\Omega(\sqrt{\log \frac{1}{\epsilon}})$ on the performance ratio of any online algorithm for any deterministic or randomized algorithm. Restricting the width, they designed an $O(\log \frac{1}{\epsilon})$-approximation algorithm.

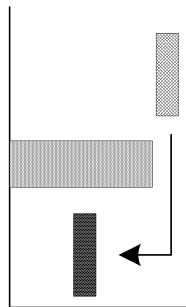


Figure 2.4: The process of packing an item in the “Tetris like” model

The oriented version of the problem was studied by Coffman, Downey and Winkler [44]. They assume a probabilistic model where item heights and widths are drawn from a uniform distribution on $[0,1]$. They show that any online algorithm which packs n items has an asymptotic expected height of at least $0.313827n$ and design an algorithm of asymptotic expected height of $0.369764n$.

2.7.3 Dynamic bin packing

A multidimensional version of a dynamic bin packing model, which was introduced in [41] for the one-dimensional case, was studied recently by Epstein and Levy [59]. This is an online model where items do not only arrive but may also leave. Each event is an arrival or a departure of an item. Durations are not known in advance, i.e., an algorithm is notified about the time that an items leaves only upon its departure. An algorithm may re-arrange the locations inside bins, but the items may not migrate between bins. In [59], the same problem was studied in multiple dimensions.

In two dimensions, they designed a 4.25-approximation algorithm for dynamical packing of squares, and provided a lower bound of 2.2307 on the performance ratio. For rectangles the upper and lower bounds are 8.5754 and 3.7 respectively. For three-dimensional cubes they presented an algorithm which is a 5.37037-approximation, and a lower bound of 2.117. For three-dimensional boxes, they supplied a 35.346-approximation algorithm and a lower bound of 4.85383. For higher dimensions, they define and analyze the algorithm NFDH for the offline box packing problem. This algorithm was studied before for rectangle packing (two-dimensional only) [43], and for square and cube packing for any dimension [126, 109], but not for box packing. For d-dimensional boxes they provided an upper bound of $2 \cdot 3.5^d$ and a lower bound of $d + 1$. Note that, as already mentioned in this survey, the best bound known for the regular offline multi-dimensional box packing problem is exponential as well. For d-dimensional cubes they provided an upper bound of $O\left(\frac{d}{\ln d}\right)$ and a lower bound of 2.

One older paper by Coffman and Gilbert [45] studies a related problem. In this problem, squares of a bounded size, which arrive and leave at various times, must be kept in a single bin. The paper gives lower bounds on the size of such a bin, so that all squares can fit. It is not allowed to re-arrange the locations in the bin.

2.7.4 Packing rectangles in a single rectangle

Another version is concerned with maximizing the number, area, or weight of a subset of the input rectangles, that can be packed into a larger rectangle (of given height and width). The maximization problem with respect to the number of rectangles was studied already in 1983 by Baker et al. [11]. They designed an asymptotic $\frac{4}{3}$-approximation. This offline problem was recently studied by Jansen and Zhang [96, 95]. The first paper considered the case of weighted rectangles, and maximizing the total weight packed, whereas the second one considered unweighted rectangles, and maximizing the number of packed rectangles. The problem is considered without rotation.

In [96], Jansen and Zhang proved that there exists an asymptotic FPTAS, and an absolute PTAS, for packing squares into a rectangle. For rectangles they gave an approximation algorithm with asymptotic ratio of at most two, and a simple one with an absolute ratio of $2 + \varepsilon$. In [95], Jansen and Zhang gave a more complicated algorithm for the weighted case with an absolute ratio of $2 + \varepsilon$. This algorithm has higher running time than the one for the unweighted problem. A special case of weights is simply the area of rectangles. The area maximization problem was studied by Caprara and Monaci [31]. They designed an algorithm with (absolute) approximation

ratio $3 + \varepsilon$.

An online version was studied by Han, Iwama and Zhang [87]. In this version, we are given a unit square bin, rectangles arrive online, and the algorithm needs to decide whether to accept an arriving rectangle or not. The goal is again to maximize the packed area. They showed that if the algorithm is not allowed to remove rectangles accepted in the past, no algorithm with constant approximation ratio exists. This holds already for squares. It is easy to see that this holds with the following example. Take a first square which is very small, and another one which fills the bin completely. An algorithm must accept the first square and therefore cannot accept the larger one. Next, they show that there is no algorithm with constant approximation ratio exists for rectangles, even if the algorithm is allowed to remove previously accepted rectangles. Therefore, the paper studies removable square packing. Before describing the results, we discuss a related paper which was used in this paper.

Januszewski and Lassak [97] studied a similar problem from the point of view of finding a threshold $\alpha \le 1$ such that a set of squares of total area of at most α can be always packed online in a bin, without re-arranging the contents of the bin. They showed that $\frac{5}{16}$ is a lower bound on α. Moreover, the considered this problem for multidimensional cubes, and showed a lower bound of $\frac{1}{2^d-1}$ for $d \ge 5$. For the packing they used a nice tool which they called bricks. A brick is a rectangle, where the ratio of the maximum between height and width to the minimum between the two remains the same after cutting the rectangle into two identical parts. Clearly, this can work if the ratio is $\sqrt{2}$.

Han, Iwama and Zhang [87] adopted this method. They showed that any algorithm has performance ratio of at least $\phi + 1 \approx 2.618$. They designed a matching algorithm for the case where re-arranging is allowed, and a 3-approximation algorithm without re-arranging. A direct consequence is that a lower bound on α for two dimensions is $\frac{1}{3}$.

Finally, another related problem is packing squares or rectangles into a square or rectangle of minimum size, where arbitrary rotations are allowed (not just over $90°$). For example, five unit squares can be packed inside a square with side $2 + \frac{1}{2}\sqrt{2}$, by placing four squares in the corners and one in the center at a $45°$ angle. For a survey on packing equal squares into a square, see [72]. Novotný [134] showed that any set of squares with total area 1 can be packed in a rectangle of area at most 1.53 (without rotations).

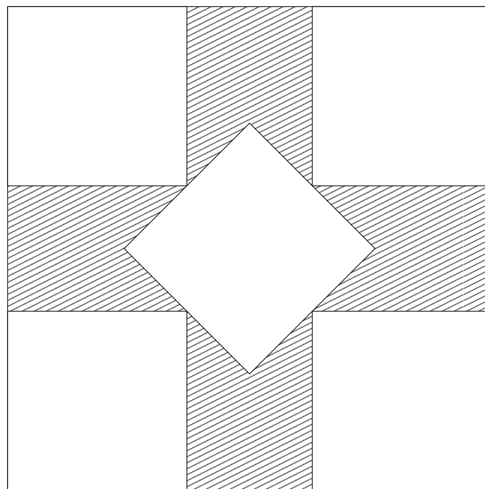


Figure 2.5: The optimal packing for five unit squares

Chapter 3

An approximation algorithm for square packing

Square packing is a special case of two-dimensional bin packing where all the input items are squares. These items need to be packed into bins which are unit squares using only orthogonal packings. The items must be assigned positions in such a way that no two items in the same bin overlap. The goal is to minimize the number of bins used.

Most of the previous work on bin packing has focused on the *asymptotic performance ratio* (approximation ratio), where the focus is on the long-term behavior of algorithms. In contrast, in the current chapter we consider the *absolute* approximation ratio [138, 150].

Attaining an absolute approximation ratio of R is more difficult than attaining an asymptotic approximation ratio of R, because in the second case an algorithm is allowed to “waste” a constant number of bins, which allows e.g. the classification of items followed by a packing where each class is packed separately.

Leung et al. [115] showed that it is NP-hard to determine whether or not a given set of squares can be packed in a single bin. This implies that there cannot be a polynomial-time algorithm with an absolute approximation ratio less than 2, unless P $=$ NP. Such an algorithm could be used to determine (in polynomial time) whether a set of squares fits into a single bin: for a given set of items, if that algorithm packs them into two bins, they cannot be packed in a single bin because the absolute approximation ratio is strictly less than 2.

We now present an algorithm for square packing with an absolute approximation ratio of 2, which is optimal provided P $\neq$ NP.

3.1 Subroutines for the algorithm

We define the *size* of a square p, denoted by $s(p)$, as the length of one of its edges. We classify items according to their size. *Huge* items have size greater than $2/3$. *Big* items have size in $(1/2, 2/3]$. *Medium-sized* items have size in $(1/3, 1/2]$. Finally, *small* items have size at most $1/3$.

All the non-small items will be packed using the algorithm FIRST FIT DECREASING SIZE

(FFDS). This algorithm works as follows. First, huge and big items are packed into bins: one item per bin, in order of increasing size. Each item is placed in a corner of its bin. This gives a list B of bins. Next, the medium-sized items are sorted in order of decreasing size, giving a list L'.

The algorithm now does the following repeatedly. It checks whether the first three items of L' can be packed together with the first bin in B, i.e. the one that contains the smallest big item. If the three items fit there, they are placed there; otherwise the first four items from L' are put in a new, empty bin. The packed items are then removed from L', the first bin is removed from B if it was used to pack them, and the algorithm continues in the same way. If $B = \emptyset$ at some point, the remaining items in L' are packed four to a bin in new bins, until all items are packed.

Ferreira, Miyazawa, and Wakabayashi [68] define FFDS and prove the following.

Lemma 3.1 (Ferreira, Miyazawa, Wakabayashi [68]) *Let L be a list of squares that all have size greater than $1/3$. Then the algorithm FFDS applied to L generates a packing where each bin, except possibly one, contains*

- *One big or huge item and no medium-sized items, or*
- *One big item and three medium-sized items, or*
- *Four medium-sized items*

The remaining bin, if there is one, contains at most three items, including at most one big item. The packing that FFDS generates is an optimal packing for L.

To pack the small items, we will use the algorithm NEXT FIT DECREASING (NFD) as a subroutine. The version of the algorithm considered here packs squares into a rectangle of size $a \times b$. The idea of this algorithm is very simple. First we sort the squares into non-increasing order. We pack items into slices. The width of a slice is b. We use NEXT FIT on the sorted list of items, considering the slices as bins. When a new slice is allocated, its height is set equal to the height of the first item placed in it. Since the items are packed in order of non-increasing size, subsequent items fit in the slice. Slices are allocated from the rectangle going from bottom to top. The algorithm halts either when all items are packed, or when it is impossible to allocate a slice. In the later case, some items remain unpacked.

Meir and Moser [126] introduce NEXT FIT DECREASING and prove the following Lemma:

Lemma 3.2 (Meir & Moser [126]) *Let L be a list of squares with sides $x_1 \geq x_2 \geq \ldots$ Then L can be packed in a rectangle of height $a \geq x_1$ and width $b \geq x_1$ using* NEXT FIT DECREASING *if one of the following conditions is satisfied:*

- *the total area of items in L is at most $x_1^2 + (a - x_1)(b - x_1)$.*
- *the total area of items in L is at most $ab/2$.*

3.2 Algorithm

In this section, we give a detailed description of the algorithm. We start by applying the algorithm FFDS from [68] to the items of size greater than $1/3$. After this, only the small items remain to be packed. These items are packed in three steps. If at some point during these three steps, all small items are already packed, the algorithm halts.

1. Bins containing only a big item and no medium-sized items are filled further with small items.
2. A bin containing at most three medium-sized items, or a big item and at most two medium-sized items, is used if it exists. There can be at most one such bin by Lemma 3.1.
3. Finally, if there are still small items left, they are packed into bins by themselves.

The details of these three steps are described below.

Step 1 *Bins with one big item, but no medium-sized items.* The big item is placed into the lower left corner of the bin. Denote its size by x. The remaining area can be divided into two rectangles, one of dimensions 1 by $1-x$ at the top of the bin and one of dimensions $1-x$ by x next to the big item. Use Next Fit Decreasing to pack the first rectangle. Continue until an item can no longer be placed, place that item in the second rectangle. Note that this is possible since the big item has size at most $2/3$ and small items have size at most $1/3$.

Step 2A *A bin with only one medium-sized item.* Pack items as in Step 1, the medium-sized item in the lower left corner.

Step 2B *A bin with two items, at least one medium-sized.* Place the largest item, of size x_1, in the lower left corner of the bin. Place the second largest item next to it, aligned with the bottom of the bin and as far to the left as possible. There is an unoccupied region of dimensions 1 by $1-x_1$ at the top of the bin. Pack small items into this region using NFD.

Step 2C *A bin with three items, at least two medium-sized.* Place the first two items as in Step 2B. Place the third item, of size x_3, on top of the first one, aligned with the left edge of the bin and as far down as possible. This leaves an unoccupied region of dimensions $1-x_3$ by $1-x_1$ in the top right corner of the bin. Pack small items into this region using NFD.

Step 3 *Bins with only small items.* Pack items into new bins using NFD, opening a new bin whenever items can no longer be placed in the current bin.

3.3 Approximation ratio

Lemma 3.3 *If there are any unpacked small items left after Step 2, each bin that is packed so far contains a total area of at least* $4/9$.

Proof The lemma clearly holds for any bins with huge items.

Bins that are packed in Step 1 or 2 are packed exactly as in the proof of Lemma 4.3 in [141]. Step 1 and 2A correspond to Case 2 from that proof, Step 2B corresponds to Case 4 and 2C corresponds to Case 5.

It follows immediately from that proof that in all cases, the used area is at least $4/9$. □

Lemma 3.4 *Consider a bin that is packed in Step 3. If after packing this bin, there are still unpacked small items left, it contains a total area of at least* $9/16$.

Proof We distinguish between cases. Since all items have size at most $1/3$, at least nine items can be packed together in the bin, and NFD allocates at least three slices.

Case 1 *The first slice contains at least four items.*

Denote the size of the largest item by x, the size of the largest item in the second slice by y and the size of the first item that can no longer be placed by z. Denote the total area of items starting from the second slice by f. By Lemma 3.2, we have $f + z^2 > y^2 + (1-y)(1-x-y)$. Therefore in the entire bin we pack at least

$$x^2 + 4y^2 + (1-y)(1-x-y) - z^2 \geq x^2 + 3y^2 + (1-y)(1-x-y) = g.$$

We have $\frac{\partial g}{\partial x} = 2x + y - 1$, which is negative for $0 \leq x < 1/3$ and $0 \leq y \leq 1/3$. We find that g has a minimum of $29/48 > 9/16$ for $\{(x,y) \in \mathbb{R}^2 | 0 \leq x \leq 1/3, 0 \leq y \leq 1/3\}$, which is attained for $x = 1/3$ and $y = 5/24$.

In the remaining cases, the four largest items do not fit next to each other in a bin.

Case 2 *The first slice contains three items, but the second slice contains at least four items.*

Denote the sizes of the first items in the first three slices by x, y and z, respectively. Denote the sizes of the other items in the first slice by x_1, x_2. Again using Lemma 3.2, we pack at least

$$x^2 + x_1^2 + x_2^2 + y^2 + 3z^2 + (1-z)(1-x-y-z).$$

We are interested in its minimum under the conditions that $0 < z \leq y \leq x_2 \leq x_1 \leq x \leq 1/3$ and $x + x_1 + x_2 + y \geq 1$. By distinguishing between the cases $y \leq 2/9$ and $y > 2/9$, we find that this function has a minimum of $743/1296 > 0.5733 > 9/16$ which is attained for $x = 1/3$, $x_1 = x_2 = y = 2/9$ and $z = 13/72$.

Case 3 *The first two slices both contain three items.*

Denote the size of the largest item by x, the size of the largest item in the second slice by y and the size of the second largest item in the second slice by z. By Lemma 3.2, any set of squares with total area at most $(1-x-y)/2$ can be packed by NFD starting from the third slice. Thus NFD packs at least $(1-x-y)/2 - z^2$ in that region, and in total at least

$$x^2 + 3y^2 + z^2 + (1-x-y)/2.$$

We have $x > 1/4$, $y > 1/4$, and $z > (1-y)/3$ since there are only three items in the first two slices. The expression is monotonically increasing in x, y and z on this domain, and we find that it is at least $9/16$ (attained for $x = y = z = 1/4$). □

Theorem 3.1 *The algorithm has an absolute approximation ratio of 2.*

Proof Denote the number of bins with a huge item by h, the number of bins that have a big item (but no medium-sized items) by b, the number of bins with medium-sized items (and possibly a big item) by m and the number of bins with (only) small items of total area at least $9/16$ by s. Our algorithm may generate one bin (the last one) that has only small items but with total area less than $9/16$. Thus the number of bins produced by the algorithm is at most

$$h + b + m + s + 1.$$

Since FFDS is an optimal algorithm, for the optimal solution OPT we find OPT $\geq h+b+m$. Thus as long as $s+1 \leq h+b+m$, our algorithm uses at most twice as many bins as an optimal solution.

Suppose $s \geq h+b+m$. First of all, if $h+b+m = 0$, then by Lemma 3.4 we have OPT $\geq \frac{9}{16}s$. If $s = 0$, then OPT $= 1$ (assuming nonzero input) and the algorithm is optimal. Otherwise, OPT $> s/2$, and therefore OPT $\geq (s+1)/2$.

Suppose $s \geq h+b+m \geq 1$. This implies that there are bins packed with only small items. In other words, we do not run out of items while packing small items in Steps 1 or 2. Lemma 3.3 guarantees that in this case, all bins packed so far contain a total area of at least $4/9$. Furthermore, all bins with only small items, except possibly the last one, have total area at least $9/16$ by Lemma 3.4.

Thus in the case that $s \geq h+b+m$, each bin except possibly the last one is on average strictly more than half full, since $4/9 + 9/16 > 1$ and $s \geq 1$. (Note that if the last bin with only small items does not contain at least an area of $9/16$, it is not counted in s.) This implies that any packing of this input requires strictly more than $(h+b+m+s)/2$ bins, and therefore at least $(h+b+m+s+1)/2$ bins. This concludes the proof. □

Chapter 4

Optimal online algorithms for multidimensional packing

This chapter is concerned with online multidimensional packing problems. Apart from the standard d-dimensional packign problem which was defined in the Introduction, we also consider the following variants:

- In the *hypercube packing* problem we have the restriction that all items are hypercubes, i.e. an item has the same size in every dimension.
- In *variable-sized* bin packing, bins of various sizes are available to be used for packing and the goal is to minimize the total size of all the bins used.
- In *resource-augmented* bin packing, the online algorithm has larger bins at its disposal than the offline algorithm, and the goal is to minimize the number of bins used.

The offline versions of these problems are NP-hard, while even with unlimited computational ability it is impossible in general to produce the best possible solution online. We consider online approximation algorithms.

The on-line one-dimensional variable-sized bin packing problem was first investigated by Friesen and Langston [73]. Csirik [49] proposed the VARIABLE HARMONIC algorithm and showed that it has performance ratio at most Π_∞. Seiden [139] showed that this algorithm is optimal among bounded space algorithms.

The on-line one-dimensional resource augmented bin packing problem was studied by Csirik and Woeginger [54]. They showed that the optimal bounded space asymptotic performance ratio is a function $\rho(b)$ of the size b of the bins of the online algorithm.

Our Results:

- We begin by presenting a bounded space algorithm for the packing of hypercubes. An interesting feature of the analysis is that although we show the algorithm is optimal, we do not know the exact asymptotic performance ratio. The asymptotic performance ratio is $\Omega(\log d)$ and $O(d/\log d)$.

- We then extend this algorithm to a bounded space algorithm for general hyperbox packing and show that this algorithm is also optimal, with an asymptotic performance ratio of $(\Pi_\infty)^d$. This solves the problem of how to pack hyperboxes using only bounded space, which had been open since 1993.
- We present a bounded space algorithm for the variable-sized multidimensional bin packing problem. As for the first algorithm above, we do not know the exact asymptotic performance ratio.
- We then give an analogous algorithm for the problem of resource augmented online bin packing. This algorithm is also optimal, and it has an asymptotic performance ratio of $\prod_{i=1}^{d} \rho(b_i)$ where $b_1 \times \cdots \times b_d$ is the size of the bins that the online algorithm uses.

We will use the well-known technique of weighting functions. This technique was originally introduced for one-dimensional bin packing algorithms [153, 98]. In [141], it was demonstrated how to use the analysis for one-dimensional algorithms to get results for higher dimensions. In contrast, in the current chapter we will define weighting functions directly for multidimensional algorithms, without using one-dimensional algorithms as subroutines.

New Technique: To construct the bounded space algorithm we adapt some of the ideas used in previous work. Specifically, the algorithm of [51] also required a scheme of partitioning bins into sub-bins, and of sub-bins into smaller and smaller sub-bins. However, in order to keep a constant number of bins active, we had to introduce a new method of classifying items. Our key improvement is that there is not one single class of "small" items like all the standard algorithms have, but instead we partition the items into an infinite number of classes that are grouped into a finite number of groups. The hypercube packing algorithm uses an easier scheme for the same purpose. This is a more direct extension of the method used in [46].

4.1 Packing hypercubes

In this section we define the algorithm for hypercubes, denoted by ALG_ε. In the next section we extend it to deal with hyperboxes. Let the *size* of hypercube p, $s(p)$ be the length of each side of the hypercube.

The algorithm has a parameter $\varepsilon > 0$. Let $M \geq 10$ be an integer parameter such that

$$M \geq 1/\left(1 - (1-\varepsilon)^{1/(d+1)}\right) - 1.$$

We distinguish between "small" hypercubes (of size smaller or equal to $1/M$) and "big" hypercubes (of size larger than $1/M$). The packing algorithm will treat them in different ways.

All *large* hypercubes are packed using a multidimensional version of HARMONIC [114]. The hypercubes are assigned a type according to their size: type i items have a size in the interval $(1/(i+1), 1/i]$ for $i = 1, \ldots, M-1$. The bins that are used to pack items of these types all contain items of only one type. We use the following algorithm to pack them. A bin is called *active* if it can still receive items, otherwise it is *closed*.

Algorithm **ASSIGNLARGE**(i) At all times, there is at most one active bin for each type. Each bin is partitioned into i^d hypercubes (sub-bins) of size $1/i$ each (the sub-bins create a grid of i strips in each dimension). Each such sub-bin can contain exactly one item of type i. On arrival of a type i item it is assigned to a free sub-bin (and placed anywhere inside this sub-bin). If all sub-bins are taken, the previous active bin is closed, a new active bin is opened and partitioned into sub-bins.

The *small* hypercubes are also assigned types depending on their size, but in a different way. Consider an item p of size $s(p) \leq 1/M$. Let k be the largest non-negative integer such that $2^k s(p) \leq 1/M$. Clearly $2^k s(p) > 1/(2M)$. Let i be the integer such that $2^k s(p) \in (1/(i+1), 1/i]$, $i \in \{M, \dots 2M-1\}$. The item is defined to be of type i. Each bin that is used to pack small items contains only small items with a given type i. Note that items of very different sizes may be packed together in one bin. We now describe the algorithm to pack a new small item of type i for $i = M, \dots, 2M-1$. A sub-bin which received a hypercube is said to be *used*. A sub-bin which is not used and not cut into smaller sub-bins is called *empty*.

Algorithm **ASSIGNSMALL**(i) The algorithm maintains a single active bin. Each bin may during its use be partitioned into sub-bins which are hypercubes of different sizes of the form $1/(2^j i)$. When an item p of type i arrives we do the following. Let k be the integer such that $2^k s(p) \in (1/(i+1), 1/i]$.

1. If there is an empty sub-bin of size $1/(2^k i)$, then the item is simply assigned there and placed anywhere within the sub-bin.

2. Else, if there is no empty sub-bin of any size $1/(2^j i)$ for $j < k$ inside the current bin, the bin is closed and a new bin is opened and partitioned into sub-bins of size $1/i$. Then the procedure in step 3 is followed, or step 1 in case $k = 0$.

3. Take an empty sub-bin of size $1/(2^j i)$ for a maximum $j < k$. Partition it into 2^d identical sub-bins (by cutting into two identical pieces, in each dimension). If the resulting sub-bins are of size larger than $1/(2^k i)$, take *one* of them and partition it in the same way. This is done until sub-bins of size $1/(2^k i)$ are reached. The new item is assigned into one such sub-bin.

Finally, the main algorithm only determines the type of newly arriving items and assigns them to the appropriate algorithms. The total number of active bins is at most $2M - 1$. In order to perform a competitive analysis, we prove the following claims.

Claim 4.1 *For a given $i \geq M$, consider an active bin of type i. At all times, the number of empty sub-bins in it of each size except $1/i$ is at most $2^d - 1$.*

Proof Note that the number of empty sub-bins of size $1/i$ decays from i^d to zero during the usage of such a bin. Consider a certain possible size r of a sub-bin in it. When a sub-bin of some size r is created, it is due to partition of a larger sub-bin. This means that there were no empty

sub-bins of size r before the partition. Afterwards, there are at most $2^d - 1$ of them for each size that has been created during the partitioning (for the smallest size into which the sub-bin is partitioned, 2^d sub-bins created, but one is immediately used). □

Claim 4.2 *For a given $i \geq M$, when a bin of type i is about to be closed, the total volume of empty sub-bins in the bin is at most $1/i^d$.*

Note that the above claims bound the volume of sub-bins that are not used at all. There is some waste of volume also due to the fact that each item does not fill its sub-bin totally. We compute this waste later.

Proof For $i \geq M$, when a bin of type i is to be closed, there are no empty sub-bins of size $1/i$ in it. There are at most $2^d - 1$ empty sub-bins of each other size by Claim 4.1. This gives a total unused volume of at most $(2^d - 1)\sum_{k \geq 1}(2^k i)^{-d} = 1/i^d$. □

Claim 4.3 *The occupied volume in each closed bin of type $i \geq M$ is at least $1 - \varepsilon$.*

Proof A hypercube which was assigned into a sub-bin of size $1/(2^k i)$ always has size of at least $1/(2^k(i+1))$. Therefore the ratio of occupied space and existing space in each used sub-bin is at least $i^d/(i+1)^d$. When a bin is closed, the total volume of used sub-bins is at least $1 - 1/i^d$ by Claim 4.2. Therefore the occupied volume in the bin is at least $i^d/(i+1)^d(1 - 1/i^d) = (i^d - 1)/(i+1)^d$. We use $i \geq M$ and $M^d \geq M + 1$ to get

$$(i^d - 1)/(i+1)^d \geq (M^d - 1)/(M+1)^d \geq (\frac{M}{M+1})^{d+1} \geq 1 - \varepsilon.$$

□

Now we are ready to analyze the performance. We define a weighting function for ALG_ε. Each item p with type $1 \leq i \leq M - 1$ has weight $w_\varepsilon(p) = 1/i^d$. Each item p' of higher type has weight $w_\varepsilon(p') = (s(p))^d/(1-\varepsilon)$ which is the volume of the item divided by $(1 - \varepsilon)$. We begin by showing that this weighting function is valid for our algorithm.

Lemma 4.1 *For all input sequences σ,*

$$\text{cost}_{\text{ALG}_\varepsilon}(\sigma) \leq \sum_{p \in \sigma} w_\varepsilon(p) + 2M - 1.$$

Proof Each closed bin of type $1 \leq i \leq M-1$ contains i^d items. All sub-bins are used when the bin is closed, and thus it contains a total weight of 1. Each closed bin of type $M \leq i \leq 2M - 1$ has occupied volume of at least $1 - \varepsilon$ by Claim 4.3, and therefore the weights of the items in such a bin sum up to at least 1. At most $2M - 1$ bins are active. Thus the total number of bins used by ALG_ε for a given input sequence σ is upper bounded by the total weight of the items plus $2M - 1$. □

By this Lemma, for any given $\varepsilon > 0$, the asymptotic performance ratio of our algorithm can be upper bounded by the maximum amount of weight that can be packed in a single bin: for a given input sequence σ (with fixed weight w), the offline algorithm minimizes the number of bins that it needs to pack all items in σ by packing as much weight as possible in each bin. If

it needs k bins, the performance ratio on this input is w/k, which is also the average weight per offline bin.

Therefore we need to find the worst case offline bin, i.e. an offline bin which is packed with a maximum amount of weight. However, for the case of cubes, we only have $M+1$ different types of items. All large items of type i have the same weight. All small items have the same ratio of weight to volume. Therefore the exact contents of a bin are not crucial. In order to define a packed bin, we only need to know how many items there are of each type, and the volume of the small items. To maximize the weight we can assume that the large items are as small as possible (without changing their type), and the rest of the bin is filled with small items.

Formally, we define a *pattern* as a tuple $q = \langle q_1, \ldots, q_{M-1} \rangle$, where there exists a feasible packing into a single bin containing q_i items of type i for all $1 \le i \le M-1$. This generalizes the definition from [140]. The weight of a pattern q is at most

$$w_\varepsilon(q) = \sum_{i=1}^{M-1} \frac{q_i}{i^d} + \frac{1}{1-\varepsilon}\left(1 - \sum_{i=1}^{M-1} \frac{q_i}{(i+1)^d}\right). \tag{4.1}$$

Note that for any given pattern the amounts of items of types $M, \ldots, 2M-1$ are unspecified. However, as mentioned above, the weight of such items is always their volume divided by $1-\varepsilon$. Therefore (4.1) gives an upper bound for the total weight that can be packed in a single bin for a given pattern q. Summarizing, we have the following Theorem.

Theorem 4.1 *The asymptotic performance ratio of* ALG_ε *is upper bounded by* $\max_q w_\varepsilon(q)$, *where the maximum is taken over all patterns q that are valid for* ALG_ε.

In order to use the Theorem, we need the following geometric Claim. We immediately formulate it in a general way so that we can also apply it in the next section.

Claim 4.4 *Given a packing of hyperboxes into bins, such that component j of each hyperbox is bounded in an interval $(1/(k_j+1), 1/k_j]$, where $k_j \ge 1$ is an integer for $j = 1, \ldots, d$, then each bin has at most $\prod_{j=1}^{d} k_j$ hyperboxes packed in it.*

Proof We prove the claim by induction on the dimension. Clearly for $d = 1$ the claim holds. To prove the claim for $d > 1$, the induction hypothesis means that a hyperplane of dimension $d-1$ through the bin which is parallel to one of the sides (the side which is the projection of the bin on the first $d-1$ dimensions) can meet at most $\prod_{j=1}^{d-1} k_j$ hyperboxes. Next, take the projection of the hyperboxes and the bin on the last axis. We get short intervals of length in $(1/(k_d+1), 1/k_d]$ (projections of hypercubes) on an main interval of length 1 (the projection of the bin). As mentioned above, each point of the main interval can have the projection of at most $\prod_{j=1}^{d-1} k_j$ items. Consider the short intervals as an interval graph. The size of the largest clique is at most $\prod_{j=1}^{d-1} k_j$. Therefore, as interval graphs are perfect, we can color the short intervals using $\prod_{j=1}^{d-1} k_j$ colors. Note that the number of intervals of each independent set is at most k_d (due to length), and so the total number of intervals is at most $\prod_{j=1}^{d} k_j$. □

Lemma 4.2 *Let*

$$\alpha = \liminf_{\varepsilon \to 0} \max_{q} w_\varepsilon(q),$$

where the maximum is taken over all patterns q that are valid for ALG$_\varepsilon$. *Then the asymptotic performance ratio of any bounded space algorithm is at least α.*

Proof We show that there is no bounded space algorithm with an asymptotic performance ratio strictly below α. For any $\varepsilon' > 0$, there exists an $\varepsilon \in (0, \varepsilon')$ such that $\mathcal{R}^\infty(\text{ALG}_\varepsilon) \leq (1+\varepsilon')\alpha$. Consider the pattern q for which $w_\varepsilon(q)$ is maximal. We write $w_\varepsilon(q) = (1+\varepsilon'')\alpha$ for some $\varepsilon'' \in [0, \varepsilon']$.

Note that a pattern does not specify the precise sizes of any of the items in it. Based on q, we define a set of hypercubes that can be packed together in a single bin. For each item of type i in q, we take a hypercube of size $1/(i+1) + \delta$ for some small $\delta > 0$. Take

$$V_\delta = 1 - \sum_{i=1}^{M-1} q_i \left(\frac{1}{i+1} + \delta \right)^d .$$

We add a large amount of small hypercubes of total volume V_δ, where the sizes of the small hypercubes are chosen in such a way that they can all be packed in a single bin together with the large hypercubes prescribed by q. By the definition of a pattern, such a packing is feasible for δ sufficiently small.

Define the following input for a bounded space algorithm. Let N be a large constant. The sequence contains M phases. The last phase contains a volume NV_δ of small hypercubes. Phase i ($1 \leq i \leq M-1$) contains Nq_i hypercubes of size $1/(i+1) + \delta$. After phase i, almost all hypercubes of this phase must be packed into closed bins (except a constant number of active bins). Each such bin may contain up to i^d items, which implies that in each phase i, $Nq_i/i^d - O(1)$ bins are closed. The last phase contributes at least $V_\delta - O(1)$ extra bins. The cost of the online algorithm is

$$\sum_{i=1}^{M-1} \frac{Nq_i}{i^d} + V_\delta - O(M).$$

But the optimal offline cost is simply N. Taking $\delta = 1/N$ and letting N grow without bound, N becomes much larger than M and the asymptotic performance ratio of any bounded space on-line algorithm is lower bounded by $\sum_{i=1}^{M-1} q_i/i^d + V_0$. Note that the weight of this set of hypercubes according to our definition of weights tends to

$$\sum_{i=1}^{M-1} \frac{q_i}{i^d} + \frac{V_0}{1-\varepsilon} = w_\varepsilon(q) = (1+\varepsilon'')\alpha$$

as $\delta \to 0$. Therefore

$$\sum_{i=1}^{M-1} \frac{q_i}{i^d} + V_0 \geq (1-\varepsilon)(1+\varepsilon'')\alpha \geq (1-\varepsilon')\alpha.$$

□

This Lemma implies that our algorithm is the best possible bounded space algorithm. More precisely, for every $\varepsilon' > 0$, there exists an $\varepsilon \in (0, \varepsilon')$ such that $\mathcal{R}^\infty(\text{ALG}_\varepsilon) \leq (1+\varepsilon')\alpha$, and no bounded space algorithm has an asymptotic performance ratio below $(1-\varepsilon')\alpha$. This also implies that our weighting function cannot be improved and determines the asymptotic performance ratio exactly. However, we have no general formula for this ratio. We do have the following bounds.

Theorem 4.2 *There exists a value of M such that the asymptotic performance ratio of ALG_ε is $O(d/\log d)$. Any bounded space algorithm (in particular ALG_ε) has an asymptotic performance ratio of $\Omega(\log d)$.*

Proof We first show the upper bound. Take $M = 2d/\log d$. The occupied area in bins of small types is at least $(\frac{M}{M+1})^{d+1}$ by the proof of Claim 4.3. This is greater than

$$\left(\frac{M+1}{M}\right)^{-d} = \left(1+\frac{1}{M}\right)^{-d} = \left(1+\frac{\log d}{2d}\right)^{-d},$$

which tends to $e^{-(\log d)/2} = (e^{\log d})^{-1/2} = 1/\sqrt{d}$ for $d \to \infty$.

Suppose the input is I. Denote by I_i the subsequence of items of type i ($i = 1, \ldots, M$), where we consider all the small types as a single type. Then we have

$$\text{ALG}(I_i) = \text{OPT}(I_i) \leq \text{OPT}(I) \text{ for } i = 1, \ldots, M-1,$$

since if items of only one type arrive, our algorithm packs them perfectly. Moreover,

$$\text{ALG}(I_M) = O(\sqrt{d}) \cdot \text{OPT}(I_M) = O(\sqrt{d}) \cdot \text{OPT}(I).$$

Thus

$$\text{ALG}(I) = \sum_{i=1}^{M} \text{ALG}(I_i) \leq (M-1)\text{OPT}(I) + O(\sqrt{d})\text{OPT}(I) = O(d/\log d)\text{OPT}(I).$$

We now prove the lower bound. Consider the following lower bound construction. (This lower bound can also be shown using the weighting function.) We use $\lceil \log d \rceil$ phases. In phase i, $N((2^i-1)^d - (2^i-2)^d)$ items of size $2^{-i}(1+\delta)$ arrive, where $\delta < 2^{-\lceil \log d \rceil} \leq 1/d$. OPT can place all these items in just N bins by using the following packing scheme. Each bin is packed identically, so we just describe the packing of a single bin. The first item is placed in a corner of the bin. We assign coordinates to the bin so that this corner is the origin and all positive axes are along edges of the bin. (The size of the bin in each dimension is 1.)

Consider any coordinate axis. We reserve the space between $(1-2^{1-i})(1+\delta)$ and $(1-2^{-i})(1+\delta)$ for items of phase i. Note that this is exactly the size of such an item. By doing this along every axis, we can place all $(2^i-1)^d - (2^i-2)^d$ items of phase i. (There would be room for $(2^i-1)^d$ items if we used all the space until $(1-2^{-i})(1+\delta)$ along each axis; we lose $(2^i-2)^d$ items because the space until $(1-2^{1-i})(1+\delta)$ is occupied.)

The minimum number of bins that any bounded space online algorithm needs to place the items of phase i is

$$N\frac{(2^i-1)^d-(2^i-2)^d}{(2^i-1)^d}=N\left(1-\left(\frac{2^i-2}{2^i-1}\right)^d\right).$$

Note that the contribution of each phase i to the total number of bins required to pack all items is strictly decreasing in i. Consider the contribution of the last phase, which is phase $\lceil\log d\rceil$. Since $\lceil\log d\rceil\le 1+\log d$, it is greater than

$$N\left(1-\left(\frac{2d-2}{2d-1}\right)^d\right)=N\left(1-\left(1-\frac{1}{2d-1}\right)^d\right)\ge N(1-e^{-1/2})>0.39N$$

for all $d\ge 2$. Thus all $\lceil\log d\rceil$ terms all contribute at least $0.39N$, and the total number of bins required is at least $0.39N(\lceil\log d\rceil)$. This implies a lower bound of $\Omega(\log d)$ on the asymptotic performance ratio of this problem. □

In [63], we give specific upper and lower bounds for dimensions $2,\ldots,7$.

4.2 Packing hyperboxes

Next we describe how to extend the algorithm for hypercubes to handle hyperboxes instead of hypercubes. This algorithm also uses the parameter ε. The value of M as a function of ε is picked so that

$$M\ge 1/\left(1-(1-\varepsilon)^{1/(d+2)}\right)-1.$$

Similarly to the previous algorithm, the hyperboxes are classified into types. An arriving hyperbox p of dimensions $(s_1(p),s_2(p),\ldots,s_d(p))$ is classified as one of $(2M-1)^d$ types depending on its components: a type of a hyperbox is the vector of the types of its components.

There are $2M-1$ types of components. A component larger than $1/M$ has type i if $1/(i+1)<s_i(p)\le 1/i$, and is called large. A component smaller than $1/M$ has type i, where $M\le i\le 2M-1$, if there exists a non-negative integer f_i such that

$$\frac{1}{i+1}<2^{f_i}s_i(p)\le\frac{1}{i}.$$

Such components are called small.

Each of the $(2M-1)^d$ types is packed separately and independently of the other types. The algorithm keeps one active bin for each type $(t_1,\ldots,t_d)$. When such a bin is opened, it is split into $\prod_{i=1}^d t_i$ identical sub-bins of dimensions $(1/t_1,\ldots,1/t_d)$. On arrival of a hyperbox h, after classification into a type, a sub-bin has to be found for it. If there is no sub-bin in the current bin that is larger than h in every dimension, we close the bin and open a new one. Otherwise, we take an empty sub-bin that has minimum volume among all sub-bins that can contain h.

Now consider the components of h one by one. If the i-th component is large, the sub-bin has the correct size in this dimension: its size is $1/t_i$ whereas the component is in $(1/(t_i+1),1/t_i]$.

If the i-th component is small, the size of the sub-bin in the i-th dimension may be too large. Suppose its size is $1/(2^{f'}t_i)$ whereas the hyperbox has size $\in (1/(2^f(t_i+1), 1/(2^f t_i)]$ in this dimension for some $f > f'$. In this case, we divide the sub-bin into two equal parts by cutting halfway (across the i-th dimension). If the new sub-bins have the proper size, take one of the two smallest sub-bins that were created, and continue with the next component. Otherwise, take one of the new sub-bins and cut it in half again, repeating until the size of a created sub-bin is $1/(2^f t_i)$.

Thus we ensure that the sub-bin that we use to pack the item h has the proper size in every dimension. We then place this item anywhere inside the sub-bin.

We now generalize the proofs from the previous section for this algorithm.

Claim 4.5 *Consider a type $(t_1,\ldots,t_d)$, and its active bin. For every vector $(f_1,\ldots,f_d) \neq 0$ of nonnegative integers such that $f_i = 0$ for each large component i, there is at most one empty sub-bin of size $(1/(2^{f_1}t_1),\ldots,1/(2^{f_d}t_d))$.*

Proof Note that the number of sub-bins of size $(1/t_1,\ldots,1/t_d)$, is initialized to be $\prod_{i=1}^d t_i$, and decays until it reaches the value zero. The cutting process does not create more than a single empty sub-bin of each size. This is true for all the sub-bins created except for the smallest size that is created in any given process. For that size we create two identical sub-bins. However, one of them is filled right away.

Furthermore, no sub-bins of existing sizes are created due to the choice of the initial sub-bin. The initial sub-bin is chosen to be of minimum volume among the ones that can contain the item, and hence all the created sub-bins (all of which can contain the item) are of smaller volume than any other existing sub-bin that can contain the item. □

Claim 4.6 *The occupied volume in each closed bin of type $(t_1,\ldots,t_d)$ is at least*

$$(1-\varepsilon)\prod_{i\in L} t_i/(t_i+1),$$

where L is the set of large components in this type.

Proof To bound the occupied volume in closed bins, note that a sub-bin which was assigned an item is full by a fraction of at least

$$\prod_{i=1}^{d}\frac{t_i}{t_i+1} \geq \left(\frac{M}{M+1}\right)^{d-|L|}\prod_{i\in L}\frac{t_i}{t_i+1}\,.$$

Considering sub-bins that were empty when the bin was closed, by Claim 4.5 there may be one empty sub-bin of each size $(1/(2^{f_1}t_1),\ldots,1/(2^{f_d}t_d))$, with the restrictions that f_i is a nonnegative integer for $i = 1,\ldots,d$, $f_i = 0$ for each large component i, and there exists some $i \in \{1,\ldots,d\}$ such that $f_i \neq 0$.

If there are no small components, there can be no empty sub-bins because large components never cause splits into sub-bins, so all sub-bins are used when the bin is closed. This gives a bound of $\prod_{i\in L} t_i/(t_i+1)$.

If there is only one small component, the total volume of all empty sub-bins that can exist is

$$\frac{1}{t_1 \dots t_d} \cdot \left(\frac{1}{2} + \frac{1}{4} + \dots \right) \leq \frac{1}{t_1 \dots t_d} \leq \frac{1}{M},$$

since one of the components is small (type is at least M) and all other components have type at least 1. The occupied volume is at least

$$\left(1 - \frac{1}{M}\right) \cdot \frac{M}{M+1} \prod_{i \in L} \frac{t_i}{t_i + 1} \geq \left(\frac{M}{M+1}\right)^{d+2} \prod_{i \in L} \frac{t_i}{t_i + 1} .$$

This holds for any $d \geq 2$ and $M \geq 2$.

If there are $r \geq 2$ small components, the total volume of empty sub-bins is at most

$$\frac{2^r - 1}{t_1 t_2 \dots t_d} \leq \frac{2^r - 1}{M^r} \leq \frac{2^r}{M^r}.$$

(We get the factor $2^r - 1$ by enumerating over all possible choices of the values f_i.) We get that the fraction of each bin that is filled is at least

$$\begin{aligned} & \left(1 - \frac{2^r}{M^r}\right) \left(\frac{M}{M+1}\right)^r \prod_{i \in L} \frac{t_i}{t_i + 1} \\ = \; & \frac{M^r - 2^r}{(M+1)^r} \prod_{i \in L} \frac{t_i}{t_i + 1} \\ \geq \; & \left(\frac{M}{M+1}\right)^{r+2} \prod_{i \in L} \frac{t_i}{t_i + 1} \\ \geq \; & \left(\frac{M}{M+1}\right)^{d+2} \prod_{i \in L} \frac{t_i}{t_i + 1}. \end{aligned}$$

The first inequality holds for $M^r - 2^r \geq M^{r+2}/(M+1)^2$, which holds for any $r \geq 2$ and $M \geq 4$. Using

$$(\frac{M}{M+1})^{d+2} \geq 1 - \varepsilon$$

we get the Claim. □

We now define a weighting function for our algorithm. The weight of a hyperbox p with components $(h_1, \dots, h_d)$ and type $(t_1, \dots, t_d)$ is defined as

$$w_\varepsilon(p) = \frac{1}{1 - \varepsilon} \prod_{i \notin L} h_i \prod_{i \in L} \frac{1}{t_i},$$

where L is the set of large components in this type.

Lemma 4.3 *For all input sequences σ,*

$$\text{cost}_{\text{alg}}(\sigma) \leq \sum_{h \in \sigma} w_\varepsilon(h) + O(1).$$

Proof In order to prove the claim, it is sufficient to show that each closed bin contains items of total weight of at least 1. Consider a bin filled with hyperboxes with type $(t_1, \ldots, t_d)$. It is sufficient to consider the subsequence σ of the input that contains only items of this type, since all types are packed independently. We build an input σ' for which both the behavior of the algorithm and the weights are the same as for σ, and show the claim holds for σ'. Let $\delta < 1/M^3$ be a very small constant.

For a hyperbox $h \in \sigma$ with components $(h_1, \ldots, h_d)$ and type $(t_1, \ldots, t_d)$, let $h' = (h'_1, \ldots, h'_d) \in \sigma'$ be defined as follows. For $i \notin L$, $h'_i = h_i$. For $i \in L$, $h'_i = 1/(t_i + 1) + \delta < 1/t_i$. As h and h' have the same type, they require a sub-bin of the same size in all dimensions. Therefore the algorithm packs σ' in the same way as it packs σ. Moreover, according to the definition of weight above, h and h' have the same weight.

Let $v(h)$ denote the volume of an item h. For $h \in \sigma$, we compute the ratio of weight and volume of the item h'. We have

$$\begin{aligned}\frac{w_\varepsilon(h')}{v(h')} &= \frac{1}{1-\varepsilon}\prod_{i\notin L} h'_i \prod_{i\in L}\frac{1}{t_i} \Big/ \prod_{i=1}^{d} h'_i \\ &= \frac{1}{1-\varepsilon}\prod_{i\in L}\frac{1}{t_i h'_i} \\ &> \frac{1}{1-\varepsilon}\prod_{i\in L}\frac{t_i+1}{t_i+M^2\delta}.\end{aligned}$$

As δ tends to zero, this bound approaches the inverse of the number in Claim 4.6. This means that the total weight of items in a closed bin is no smaller than 1. □

Just like in Section 4.1, this Lemma implies that the asymptotic performance ratio is upper bounded by the maximum amount of weight that can be packed in a single bin. We now prove a technical lemma that implies that this weighting function is also "optimal" in that it determines the true asymptotic performance ratio of our algorithm.

Definition 4.1 *The* pseudo-volume *of a hyperbox* $h = (h_1, \ldots, h_d)$ *is defined as* $\prod_{i\notin L} h_i$*, where* L *is the set of large components of* h.

Suppose that for a given set of hyperboxes X, we can partition the dimensions into two sets, S and T, such that for each dimension j in S, we have that the j-th components of all hyperboxes in X are bounded in an interval $(1/(k_j + 1), 1/k_j]$. There are no restrictions on the dimensions in T. (Thus such a partition can always be found by taking $S = \emptyset$.)

For a hyperbox $p \in X$, define the *generalized pseudo-volume* of the components in T by

$$\tilde{v}(p, T) = \prod_{j\in T} s_j(p),$$

where $s_j(p)$ is the jth component of p. Define the total generalized pseudo-volume of all hyperboxes in a set X by $\tilde{v}(X, T) = \sum_{p\in X} \tilde{v}(p, T)$.

Claim 4.7 *For a given set X of hyperboxes, for sufficiently large N, any packing of X into bins requires at least*

$$\tilde{v}(X,T)\left(1-\frac{1}{N}\right)^{|T|} \Big/ \prod_{i\in S} k_i$$

bins, where S and T form a partitioning of the dimensions as described above.

Proof We prove the claim by induction on the number of dimensions in T. For $|T| = 0$, we find that the total generalized pseudo-volume of X is simply the number of hyperboxes in X (since the empty product is 1) and thus the claim is true using Claim 4.4.

Assume the claim is true for $|T| = 0, \ldots, r-1$. Suppose $|S| = d-r < d$. Take any dimension $i \in T$. We replace each hyperbox p, with component $s_i(p)$ in dimension i, by $\lfloor N^2 h_i \rfloor$ hyperboxes that have $\frac{1}{N^2}$ as their i-th component, and are identical to p in all other components. Here N is taken sufficiently large, such that $\frac{1}{N} < h_i$. Clearly, the new input X' is no harder to pack, as we split each item into parts whose sum is smaller than or equal to the original items. The total generalized pseudo-volume of the hypercubes in X' is at most a factor of

$$1-\frac{1}{N^2 s_i(p)} \geq 1-\frac{1}{N}$$

smaller than that of X. So if we write $T' = T\backslash\{i\}$, we have

$$\tilde{v}(X',T')\cdot\frac{1}{N^2} \geq \tilde{v}(X,T)\left(1-\frac{1}{N}\right).$$

By induction, it takes at least

$$\tilde{v}(X',T')\cdot\left(1-\frac{1}{N}\right)^{r-1} \Big/ \prod_{j\in S\cup\{i\}} k_j$$

bins to pack the modified input X'. Using that $k_i = N^2$, this is

$$\tilde{v}(X,T)\cdot\left(1-\frac{1}{N}\right)^{r} \Big/ \prod_{j\in S} k_j$$

bins. □

Letting $\gamma = 1-(1-\frac{1}{N})^d$, we get that the required number is at least $\tilde{v}(X,T)(1-\gamma)/\prod_{j\in S} k_j$ bins, where $\gamma \to 0$ as $N \to \infty$. In the remainder, we will take S to be the dimensions where the components of the hyperboxes in X are large, and T the dimensions where they are small. Note that this choice of S satisfies the constraints on S above, and that this reduces the generalized pseudo-volume to the (normal) pseudo-volume defined before. We are ready to prove the following Lemma.

Lemma 4.4 *Let $\varepsilon > 0$. Suppose the maximum amount of weight that can be packed in a single bin is α_ε. Then our algorithm has an asymptotic performance ratio of α_ε, and the asymptotic performance ratio of any bounded space algorithm is at least $(1-\varepsilon)\alpha_\varepsilon$.*

Proof The first statement follows from Lemma 4.3. We show a lower bound of value which tends to α_ε on the asymptotic performance ratio of any bounded space algorithm.

Consider a packed bin for which the sum of weights is α_ε. Partition the hyperboxes of this bin into M^d types in the following way. Each component is either of a type in $\{1, \dots, M-1\}$ or small (i.e. of a type i, $i \le M$). Let N' be a large constant. The sequence consists of phases. Each phase consists of one item from the packed bin, repeated N' times. The optimal offline cost is therefore N'. Using Claim 4.7 we see that the amount of bins needed to pack a phase which consists of an item p repeated N' times is simply

$$N' w_\varepsilon(p)(1-\gamma)(1-\varepsilon).$$

Therefore the cost of an on-line algorithm is at least

$$N' \alpha_\varepsilon (1-\gamma)(1-\varepsilon) - O(1),$$

which makes the asymptotic performance ratio arbitrarily close to $(1-\varepsilon)\alpha_\varepsilon$. □

Furthermore, we can determine the asymptotic performance ratio of our algorithm for hyperbox packing. Comparing to the unbounded space algorithm in [51] we can see that all the weights we defined are smaller than or equal to the weights used in [51]. So the asymptotic performance ratio is not higher. However, it can also not be lower due to the general lower bound for bounded space algorithms. This means that both algorithms have the same asymptotic performance ratio, namely $(\Pi_\infty)^d$, where $\Pi_\infty \approx 1.691$ is the asymptotic performance ratio of the algorithm HARMONIC [114].

4.3 Variable-sized packing

In this section we consider the problem of multidimensional packing where the bins used can have different sizes. We assume that all bins are hypercubes, with sides $\alpha_1 < \alpha_2 < \ldots < \alpha_m = 1$. In fact our algorithm is more general and works for the case where the bins are hyperboxes with dimensions α_{ij} ($i = 1, \dots, m$, $j = 1, \dots, d$). We present the special case of bins that are hypercubes in this thesis in order to avoid an overburdened notation and messy technical details.

The main structure of the algorithm is identical to the one in Section 4.2. The main problem in adapting that algorithm to the current problem is selecting the right bin size to pack the items in. In the one-dimensional variable-sized bin packing problem, it is easy to see which bin will accommodate any given item the best; here it is not so obvious how to select the right bin size, since in one dimension a bin of a certain size might seem best whereas for other dimensions, other bins seem more appropriate.

We begin by defining types for hyperboxes based on their components and the available bin sizes. Once again we use a parameter ε. The value of M as a function of ε is again picked so that $M \ge 1/(1-(1-\varepsilon)^{1/(d+2)}) - 1$. An arriving hyperbox p of dimensions $(s_1(p), s_2(p), \dots, s_d(p))$ is classified as one of at most $(2mM/\alpha_1 - 1)^d$ types depending on its components: a type of a hyperbox is the vector of the types of its components. We define

$$U_i = \left\{ \frac{\alpha_i}{j} \,\middle|\, j \in \mathbb{N}, \frac{\alpha_i}{j} \ge \frac{\alpha_1}{2M} \right\}, \qquad U = \bigcup_{i=1}^{m} U_i.$$

Let the members of U be

$$1 = u_1 > u_2 > \ldots > u_{q'} = \frac{\alpha_1}{M} > \ldots > u_q = \frac{\alpha_1}{2M}.$$

The interval I_j is defined to be $(u_{j+1}, u_j]$ for $j = 1, \ldots, q'$. Note that these intervals are disjoint and that they cover $(\alpha_1/M, 1]$.

A component larger than α_1/M has type i if $s_i(p) \in I_i$, and is called large. A component smaller than α_1/M has type i, where $q' \leq i \leq q-1$, if there exists a non-negative integer f_i such that

$$u_{i+1} < 2^{f_i} s_i(p) \leq u_i.$$

Such components are called small. Thus in total there are $q - 1 \leq 2mM/\alpha_1 - 1$ component types.

Bin selection We now describe how to select a bin for a given type. Intuitively, the size of this bin is chosen in order to maximize the number of items packed relative to the area used. This is done as follows.

For a given component type t_i and bin size α_j, write $F(t_i, \alpha_j) = \max\{k \mid \alpha_j/k \geq u_{t_i}\}$. Thus for a large component, $F(t_i, \alpha_j)$ is the number of times that a component of type t_i fits in an interval of length α_j. This number is uniquely defined due to the definition of the numbers u_i. Basically, the general classification into types is too fine for any particular bin size, and we use $F(t_i, \alpha_j)$ to get a less refined classification which only considers the points u_i of the form α_j/k.

Denote by L the set of components in type $(t_1, \ldots, t_d)$ that are large. If $L = \emptyset$, we use a bin of size 1 for this type. Otherwise, we place this type in a bin of any size α_j which maximizes[1]

$$\prod_{i \in L} \frac{F(t_i, \alpha_j)}{\alpha_j}. \tag{4.2}$$

Thus we do not take small components into account in this formula. Note that for a small component, $F(t_i, \alpha_j)$ is not necessarily the same as the number of times that such a component fits into any interval of length α_j. However, it is at least M for any small component.

When such a bin is opened, it is split into $\prod_{i=1}^d F(t_i, \alpha_j)$ identical sub-bins of dimensions $(\alpha_j/F(t_1, \alpha_j), \ldots, \alpha_j/F(t_d, \alpha_j))$. These bins are then further sub-divided into sub-bins in order to place hyperboxes in "well-fitting" sub-bins, in the manner which is described in Section 4.2.

Similarly to in that section, the following claim can now be shown.

Claim 4.8 *The occupied volume in each closed bin of type* $t = (t_1, \ldots, t_d)$ *is at least*

$$V_{t,j} = (1 - \varepsilon)\alpha_j^d \prod_{i \in L} \frac{F(t_i, \alpha_j)}{F(t_i, \alpha_j) + 1},$$

where L *is the set of large components in this type and* α_j *is the bin size used to pack this type.*

[1] For the case that the bins are hyperboxes instead of hypercubes, we here get the formula $\prod_{i \in L}(F(t_i, \alpha_{ij})/\alpha_{ij})$, and similar changes throughout the text.

We now define a weighting function for our algorithm. The weight of a hyperbox p with components $(s_1(p),\ldots,s_d(p))$ and type $(t_1,\ldots,t_d)$ is defined as

$$w_\varepsilon(h) = \frac{1}{1-\varepsilon}\left(\prod_{i\notin L} s_i(p)\right)\left(\prod_{i\in L}\frac{\alpha_j}{F(t_i,\alpha_j)}\right),$$

where L is the set of large components in this type and α_j is the size of bins used to pack this type.

In order to prove that this weighting function works (gives a valid upper bound for the cost of our algorithm), we will want to modify components t_i to the smallest possible component such that $F(t_i,\alpha_j)$ does not change. (Basically, a component will be rounded to $\alpha_j/(F(t_i,\alpha_j)+1)$ plus a small constant.) However, with variable-sized bins, when we modify components in this way, the algorithm might decide to pack the new hyperbox differently. (Remember that $F(t_i,\alpha_j)$ is a "less refined" classification which does not take other bin sizes than α_j into account.) To circumvent this technical difficulty, we will show first that as long as the algorithm keeps using the same bin size for a given item, the volume guarantee still holds.

For a given type $t = (t_1,\ldots,t_d)$ and the corresponding set L and bin size α_j, define an *extended type* $\mathrm{Ext}(t_1,\ldots,t_d)$ as follows: an item p is of extended type $\mathrm{Ext}(t_1,\ldots,t_d)$ if each large component $s_i(p) \in (\frac{\alpha_j}{F(t_i,\alpha_j)+1}, \frac{\alpha_j}{F(t_i,\alpha_j)}]$ and each small component $s_i(p)$ is of type t_i.

Corollary 4.1 *Suppose items of extended type* $\mathrm{Ext}(t_1,\ldots,t_d)$ *are packed into bins of size* α_j. *Then the occupied volume in each closed bin is at least* $V_{t,j}$.

Proof In the proof of Claim 4.8, we only use that each large component $s_i(p)$ is contained in the interval $(\frac{\alpha_j}{F(t_i,\alpha_j)+1}, \frac{\alpha_j}{F(t_i,\alpha_j)}]$. Thus the proof also works for extended types. □

Lemma 4.5 *For all input sequences* σ,

$$\mathrm{cost}_{\mathrm{alg}}(\sigma) \le \sum_{h\in\sigma} w_\varepsilon(h) + O(1).$$

Proof In order to prove the claim, it is sufficient to show that each closed bin of size α_j contains items of total weight of at least α_j^d. Consider a bin of this size filled with hyperboxes of type $(t_1,\ldots,t_d)$. It is sufficient to consider the subsequence σ of the input that contains only items of this type, since all types are packed independently. This subsequence only uses bins of size α_j so we may assume that *no other sizes of bins are given*. We build an input σ' for which both the behavior of the algorithm and the weights are the same as for σ, and show the claim holds for σ'. Let $\delta < 1/M^3$ be a very small constant.

For a hyperbox $p \in \sigma$ with components $(s_1(p),\ldots,s_d(p))$ and type $t = (t_1,\ldots,t_d)$, let $p' = (s'_1(p),\ldots,s'_d(p)) \in \sigma'$ be defined as follows. For $i \notin L$, $s'_i(p) = s_i(p)$. For $i \in L$,

$$s'_i(p) = \frac{\alpha_j}{F(t_i,\alpha_j)+1} + \delta < \frac{\alpha_j}{F(t_i,\alpha_j)}.$$

Note that p' is of extended type $\mathrm{Ext}(t_1,\ldots,t_d)$. Since only one size of bin is given, the algorithm packs σ' in the same way as it packs σ. Moreover, according to the definition of weight above, p and p' have the same weight.

Let $v(p)$ denote the volume of an item p. For $p \in \sigma$, we compute the ratio of weight and volume of the item p'. We have

$$\begin{aligned}\frac{w_\varepsilon(h')}{v(p')} = \frac{w_\varepsilon(h)}{v(p')} &= \frac{1}{1-\varepsilon}\left(\prod_{i\notin L} s_i'(p)\right)\left(\prod_{i\in L}\frac{\alpha_j}{F(t_i,\alpha_j)}\right)\Bigg/\prod_{i=1}^{d} s_i'(p)\\ &= \frac{1}{1-\varepsilon}\prod_{i\in L}\frac{\alpha_j}{F(t_i,\alpha_j)s_i'(p)}\\ &> \frac{1}{1-\varepsilon}\prod_{i\in L}\frac{F(t_i,\alpha_j)+1}{F(t_i,\alpha_j)+M\frac{\alpha_j}{\alpha_1}\delta}.\end{aligned}$$

Here we have used in the last step that a component with a large type fits less than M times in a (one-dimensional) bin of size α_1, and therefore less than $M\frac{\alpha_j}{\alpha_1}$ times in a bin of size $\alpha_j \geq \alpha_1$. As δ tends to zero, this bound approaches $\alpha_j^d/V_{t,j}$. We find

$$w_\varepsilon(p) \geq \alpha_j^d \frac{v(p')}{V_{t,j}} \qquad \text{for all } p \in \sigma.$$

Then Corollary 4.1 implies that the total weight of items in a closed bin of size α_j is no smaller than α_j^d, which is the cost of such a bin. □

Suppose the optimal solution for a given input sequence σ uses n_j bins of size α_j. Denote the ith bin of size α_j by $B_{i,j}$. Then

$$\frac{\sum_{h\in\sigma} w_\varepsilon(p)}{\sum_{j=1}^m \alpha_j^d n_j} = \frac{\sum_{j=1}^m\sum_{i=1}^{n_j}\sum_{h\in B_{i,j}} w_\varepsilon(p)}{\sum_{j=1}^m \alpha_j^d n_j} = \frac{\sum_{j=1}^m\sum_{i=1}^{n_j}\sum_{h\in B_{i,j}} w_\varepsilon(p)}{\sum_{j=1}^m\sum_{i=1}^{n_j} \alpha_j^d}.$$

This implies that the asymptotic performance ratio is upper bounded by

$$\max_j \max_{X_j} \sum_{p\in X_j} w_\varepsilon(p)/\alpha_j^d, \tag{4.3}$$

where the second maximum is taken over all sets X_j that can be packed in a bin of size α_j. Similarly to Section 4.2, it can now be shown that this weighting function is also "optimal" in that it determines the true asymptotic performance ratio of our algorithm.

In particular, it can be shown that packing a set of hyperboxes X that have the same type vectors of large and small dimensions takes at least

$$\sum_{p\in X}\prod_{i\notin L}\frac{p_i}{\alpha_j}\Bigg/\prod_{i\in L}F(t_i,\alpha_j)$$

bins of size α_j, where $s_i(p)$ is the ith component of hyperbox p, t_i is the type of the ith component, and L is the set of large components (for all the hyperboxes in X). Since the cost of such a bin is α_j^d, this means that the total cost to pack N' copies of some item p is at least $N' w_\varepsilon(p)(1-\varepsilon)$

when bins of this size are used. However, it is clear that using bins of another size α_k does not help: packing N' copies of p into such bins would give a total cost of

$$N' \left(\prod_{i \notin L} s_i(p) \right) \left(\prod_{i \in L} \frac{\alpha_k}{F(t_i, \alpha_k)} \right) .$$

Since α_j was chosen to maximize $\prod_{i \in L}(F(t_i, \alpha_j)/\alpha_j)$, this expression cannot be less than $N' w_\varepsilon(p)(1-\varepsilon)$. More precisely, any bins that are not of size α_j can be replaced by the appropriate number of bins of size α_j without increasing the total cost by more than 1 (it can increase by 1 due to rounding).

This implies that our algorithm is optimal among online bounded space algorithms.

4.4 Resource augmented packing

The resource augmented problem is now relatively simple to solve. In this case, the online algorithm has bins at its disposal that are hypercubes of dimensions $b_1 \times b_2 \times \ldots \times b_d$. We can use the algorithm from Section 4.2 with the following modification: the types for dimension j are not based on intervals of the form $(1/(i+1), 1/i]$ but rather intervals of the form $(b_j/(i+1), b_j/i]$.

Then, to pack items of type $(t_1, \ldots, t_d)$, a bin is split into $\prod_{i=1}^d t_i$ identical sub-bins of dimensions $(b_1/t_1, \ldots, b_d/t_d)$, and then subdivided further as necessary.

We now find that each closed bin of type $(t_1, \ldots, t_d)$ is full by at least

$$(1-\varepsilon) B \prod_{i \in L} \frac{t_i}{t_i + 1},$$

where L is the set of large components in this type, and $B = \prod_{j=1}^d b_i$ is the volume of the bins of the online algorithm.

We now define the weight of a hyperbox p with components $(s_1(p), \ldots, s_d(p))$ and type $(t_1, \ldots, t_d)$ as

$$w_\varepsilon(p) = \frac{1}{1-\varepsilon} \left(\prod_{i \notin L} \frac{h_i}{b_i} \right) \left(\prod_{i \in L} \frac{1}{t_i} \right),$$

where L is the set of large components in this type.

This can be shown to be valid similarly as before, and it can also be shown that items can not be packed better. However, in this case we are additionally able to give explicit bounds for the asymptotic performance ratio.

4.4.1 The asymptotic performance ratio

Csirik and Woeginger [54] showed the following for the one-dimensional case.

For a given bin size b, define an infinite sequence $U(b) = \{u_1, u_2, \ldots\}$ of positive integers as follows:

$$u_1 = \lfloor 1 + b \rfloor \qquad \text{and} \qquad r_1 = \frac{1}{b} - \frac{1}{u_1},$$

and for $i = 1, 2, \ldots$

$$u_{i+1} = \lfloor 1 + \frac{1}{r_i} \rfloor \qquad \text{and} \qquad r_{i+1} = r_i - \frac{1}{u_{i+1}}.$$

Define

$$\rho(b) = \sum_{i=1}^{\infty} \frac{1}{u_i - 1}.$$

Lemma 4.6 *For every bin size $b \geq 1$, there exist online bounded space bin packing algorithms with worst case performance arbitrarily close to $\rho(b)$. For every bin size $b \geq 1$, the bound $\rho(b)$ cannot be beaten by any online bounded space bin packing algorithm.*

The following lemma was proved in Csirik and Van Vliet [51] for a specific weighting function which is independent of the dimension, and is similar to a result of Li and Cheng [116]. However, the proof holds for any positive one-dimensional weighting function w. We extend it for the case where the weighting function depends on the dimension. For a one-dimensional weighting function w_j and an input sequence σ, define $w_j(\sigma) = \sum_{p \in \sigma} w_j(p)$. Furthermore define $W_j = \sup_\sigma w_j(\sigma)$, where the supremum is taken over all sequences that can be packed into a one-dimensional bin.

Lemma 4.7 *Let σ be a list of d-dimensional rectangles, and let Q be a packing which packs these rectangles into a d-dimensional unit cube. Let w_j ($j = 1, \ldots, d$) be arbitrary one-dimensional weighting functions. For each $p \in \sigma$, we define a new hyperbox p' as follows: $s_j(p') = w_j(s_j(p))$ for $1 \leq j \leq d$. Denote the resulting list of hyperboxes by σ'. Then there exists a packing Q' which packs σ' into a cube of size $(W_1, \ldots, W_d)$.*

Proof We use a construction analogous to the one in [51]. We transform the packing $Q = Q^0$ of σ into a packing Q^d of σ' in a cube of the desired dimensions. This is done in d steps, one for each dimension. Denote the coordinates of item p in packing Q^i by $(x_1^i(p), \ldots, x_d^i(p))$, and its dimensions by $(s_1^i(p), \ldots, s_d^i(p))$.

In step i, the coordinates as well as the sizes in dimension i are adjusted as follows. First we adjust the sizes and set $s_i^i(p) = w_i(s_i(p))$ for every item p, leaving other dimensions unchanged.

To adjust coordinates, for each item p in packing Q^{i-1} we find the "left-touching" items, which is the set of items g which overlap with p in $d-1$ dimensions, and for which $x_i^{i-1}(g) + s_i^{i-1}(g) = x_i^{i-1}(p)$. We may assume that for each item p, there is either a left-touching item or $x_i^{i-1}(p) = 0$.

Then, for each item p that has no left-touching items, we set $x_i^i(p) = 0$. For all other items p, starting with the ones with smallest i-coordinate, we make the i-coordinate equal to $\max(x_i^i(g) + s_i^i(g))$, where the maximum is taken over the left-touching items of p in packing S^{i-1}. Note that we use the new coordinates and sizes of left-touching items in this construction, and that this creates a packing without overlap.

If in any step i the items need more than W_i room, this implies a chain of left-touching items with total size less than 1 but total weight more than W_i. From this we can find a set of one-dimensional items that fit in a bin but have total weight more than W_i (using weighting function w_i), which is a contradiction. □

As in [51], this implies immediately that the total weight that can be packed into a unit-sized bin is upper bounded by $\prod_{i=1}^{d} W_i$, which in the present case is $\prod_{i=1}^{d} \rho(b_i)$. Moreover, by extending the lower bound from [54] to d dimensions exactly as in [51], it can be seen that the asymptotic performance ratio of any online bounded space bin packing algorithm can also not be lower than $\prod_{i=1}^{d} \rho(b_i)$.

4.5 Conclusions

An open question left by this chapter is what the asymptotic performance ratio of the bounded space hypercube packing problem is. We can show that it is $\Omega(\log d)$, and we conjecture that it is $\Theta(\log d)$. Giving a explicit expression for the competitive ratio in variable sized packing (as a function of the bin sizes) would be harder. Already in [139] where an optimal one-dimensional bounded space algorithm was given for the variable sized problem, its ratio is unknown. It is interesting to find out whether in the multidimensional case the worst case occurs when only unit sized bins are available.

Chapter 5

Packing with rotations

In this chapter, we consider several two- and three-dimensional packing problems with rotations. This is also known as “packing of non-oriented items” [55]. Usually oriented items are considered, but in many applications, there is no reason to exclude the option of changing the orientation of items before assignment. Some applications may allow rotation only in certain directions.

The two-dimensional bin packing problem with rotations is defined as follows. The bins are unit squares and the items are rectangles that may be rotated by 90° (so their sides are still aligned with sides of the bin). In the strip packing version, the strip is two-dimensional, with a base of width one and infinite height. In the three-dimensional bin packing problem with rotations, the bins are three-dimensional cubes, and the items are non-oriented three-dimensional boxes, rotatable by 90° in all possible directions. In the strip packing version we pack these items into a three-dimensional strip with a base which is a unit square, and again, infinite height.

In three dimensions, we can also consider the case of items that may be rotated so that the width and length are interchanged, however the height is fixed. We call this problem “This Side up”, as it has applications in packing of fragile objects, where a certain face of the box must be placed on top. This three-dimensional problem, as the rotatable problems has two versions: packing into a three-dimensional strip (also called “the z-oriented 3-D packing problem” [129]) and packing into three-dimensional bins. We reserve the name This Side Up for the strip packing version. Miyazawa and Wakabayashi [131] consider several problems with rotatable items and give an upper bound of 2.64 for the This Side Up problem, which was also considered by Li and Chen [120].

In all problems, the dimensions of the items to be packed are all at most 1. Miyazawa and Wakabayashi [129] demonstrate a reduction from the general three-dimensional strip packing problem with rotations to the This Side Up problem in a strip, but this reduction does not hold for the case considered in this paper, where the three-dimensional strip always has a square base of side 1.

We improve upon the best known results for all the above problems.

- A simple and fast algorithm of asymptotic performance ratio $3/2 = 1.5$ for two-dimensional rotatable strip packing. This improves on the bound 1.613 from [131]. Additionally, we give an asymptotic approximation scheme for this problem.

- An algorithm of asymptotic performance ratio $9/4 = 2.25$ for two-dimensional rotatable packing into bins. This improves on the on-line algorithm from [58] that has an asymptotic performance ratio of slightly less than 2.45. Although this algorithm basically consists of many (easy) cases, it has the advantage that it can easily be adapted to the more complex problems listed below. This is the main reason for including this algorithm here; the result itself can be improved as a consequence of our results for strip packing.

- An algorithm which combines methods of the two above algorithms and has asymptotic performance ratio $9/4 = 2.25$ for the "This side up" problem in a strip. This improves the bound of [131] which is 2.64.

- An adaptation of the previous algorithm to packing of rotatable items in a three-dimensional strip, with the same asymptotic performance ratio $9/4 = 2.25$. This improves the bound of [131] which is 2.76.

- A simple adaptation of the two previous algorithms for the bin packing versions of these problems, with asymptotic performance ratio $9/2 = 4.5$. This improves the bound of [131] for three-dimensional bin packing of rotatable items, which is 4.89.

5.1 Strip packing

We begin by giving a simple algorithm with an upper bound of $3/2$ in Section 5.1.1. This improves on the bound 1.613 from [131]. Our main result is the following. We denote by $\text{OPT}(L)$ the height of the optimal strip packing of L which is a list of rotatable items.

Theorem 5.1 *There is an algorithm* ALG *which, given a list L of n rectangles with side lengths at most 1 that can be rotated $90°$, and a positive number $\varepsilon < 1/2$, produces a packing of L in a strip of width 1 and height $\text{ALG}(L)$ such that*

$$\text{ALG}(L) \leq (1+\varepsilon)\text{OPT}(L) + O\left(\frac{1}{\varepsilon^2}\right).$$

The time complexity of ALG *is polynomial in n.*

We use the algorithm from [107] as a subroutine. Basically we do a preprocessing step to obtain a modified list of items with only a constant number of different widths and heights. Then the number of orientations for this modified set of items is polynomial in n. For any fixed orientation, we can apply the algorithm from [107]. The crucial point of the proof is to show that we do not lose too much in the preprocessing step, and therefore have a good approximation. The additive constant for this APTAS is $O(1/\varepsilon^3)$. It is possible to get an additive constant of $O(1/\varepsilon^2)$ also in this case, but at the cost of a significantly more complex preprocessing step. The rounding procedure as presented here is due to Claire Kenyon [106].

These results immediately imply an asymptotic $(2+\varepsilon)$-approximation algorithm for two-dimensional bin packing with rotations (that runs also in time polynomial in $1/\varepsilon$).

We already described some of the methods used in [107] and [43] which are also important in the current chapter in Chapter 2, Section 2.2.2. We then first give a simple and fast $3/2$-approximation in Subsection 5.1.1. We describe and analyze our APTAS in Subsection 5.1.2

5.1.1 A $3/2$-approximation algorithm

In the next subsection, we will give an asymptotic approximation scheme for this problem. However, we believe that the algorithm presented in the current section is of independent interest; ideas from it are also used in the algorithm for the "This side up" problem.

Our algorithm places big items in reverse orientation and all other items in standard orientation.

1. The big items are stacked at the bottom of the strip, in order of decreasing width and aligned with the left side of the strip. Denote the height needed for this packing by h_1.
2. Denote the height at which the first item of width at most $2/3$ is packed by h_1' ($0 \le h_1' \le h_1$). If $h_1' < h_1$, define a substrip of width $1/3$ that starts at height h_1', at the right side of the strip. Pack items that have widths in $(0, 1/6]$ inside this strip using FFDH, until all these items are packed or until the next item to be packed would be placed (partially) above height h_1.
3. If all items that have width in $(0, 1/6]$ have now been packed:
   (a) Stack items of widths in $(1/6, 1/2]$ at the right side of the strip, on top of the substrip from step 2. Place these items in order of *increasing width*. Each item is placed as low as possible, at the extreme right of the strip, under the constraint that it does not overlap with previously placed items. Do this until all such items are packed or the next item to be packed would be placed (partially) above height h_1.
   (b) Place the unpacked items of width in $(1/3, 1/2]$ in two stacks starting at height h_1 by each time adding an item to the shortest stack. Pack the unpacked items of width in $(1/6, 1/3]$ using FFDH.
4. Else, place all remaining items above height h_1 using the algorithm FFDH.

Theorem 5.2 *For this algorithm, we have* $\text{ALG}_1(L) \le \frac{3}{2}\text{OPT}(L) + 3$ *for any input list* L.

Proof We start with the simple inequality. We have

$$\text{OPT}(L) \ge h_1 \tag{5.1}$$

because the packing in step 1 is optimal for the big items. If no items are packed in step 3(b) or 4 (all items are already packed) then it follows that the algorithm gives an optimal packing.

Denote the list of items packed in step i by L_i, and the height of that packing by h_i. By Theorem 2.2, for Step 4 we have $h_4 \le \frac{3}{2}A(L_4) + 1$ and therefore

$$A(L_4) \ge \frac{2}{3}(h_4 - 1). \tag{5.2}$$

Denote the height of the substrip in step 2 by h_2 (the top of the substrip is at height h_1 or at the top of the highest item placed inside it), and the list of items packed in it by L_2. Note that all items placed inside this substrip have width at most half the width of the substrip.

Suppose we multiply all the widths of items in the substrip by 3, as well as the width of the substrip itself. Denote the new area of the list L_2 by $A'(L_2)$. Then Theorem 2.2 implies $h_2 \leq \frac{3}{2}A'(L_2) + 1$. Since $A'(L_2) = 3A(L_2)$, we can conclude

$$A(L_2) \geq \frac{2}{9}(h_2 - 1). \tag{5.3}$$

Case 1 No items are packed in Step 3.

The only interesting case is where some items are packed in Step 4. Consider Step 2. Below the substrip, a width of at least $2/3$ is covered everywhere by packed items. Next to the substrip, a width of at least $1/2$ is covered, and in the substrip, we have (5.3). On this part of the strip we have therefore packed items of total area at least

$$\frac{1}{2}h_2 + \frac{2}{9}(h_2 - 1) \geq \frac{2}{3}(h_2 - 1).$$

We can conclude that on the main strip, we have packed items of total area at least $\frac{2}{3}(h_1 - 2)$ below a height of h_1, since $h_2 \geq h_1 - h_1' - 1$ in the present case. In summary, we have that our algorithm packs items to a height of $h_1 + h_4$, and the area of these items is at least $\frac{2}{3}(h_1 - 2 + h_4 - 1)$ by (5.2). This implies that

$$\text{OPT}(L) \geq \frac{2}{3}(h_1 + h_4 - 3)$$

and therefore

$$\text{ALG}_1(L) \leq \frac{3}{2}\text{OPT}(L) + 3.$$

Case 2 Some items are packed in Step 3. Denote the height of the packing in Step 3(b) by h_3 (i.e. the extra height that is packed above h_1). We have $\text{ALG}_1(L) = h_1 + h_3$.

In Step 3(a), some item(s) may not always be placed directly on top of the previous item. This happens if they would overlap with an item from Step 1. Suppose each item placed in Step 3(a) is placed directly on top of the previous item in that Step. Then a width of $1/2 + 1/6 = 2/3$ is everywhere covered by items above h_1' and below $h_1 - 1$. Below h_1', a width of $2/3$ is covered everywhere.

Above h_1, a width of at least $2/3$ is covered in the part of the strip that is used in Step 3(b), apart from at most two parts of total height at most 2. We find

$$\text{OPT}(L) \geq \frac{2}{3}(h_1 - 1 + h_3 - 2)$$

and therefore

$$\text{ALG}_1(L) = h_1 + h_3 \leq \frac{3}{2}\text{OPT}(L) + 3.$$

Now suppose there is an item placed in Step 3(a) that is not placed on top of its predecessor, or the first item in Step 3(b) would have been placed in Step 3(a) if its width had been smaller (i.e. it is placed in 3(b) because of its width, not because of its height). Denote the width of the *last* such item by w, and the height at which this item is placed by $h_3' \in [h_1', h_1]$. Then up to height $\min(h_3', h_1 - 1)$, a width of at least $1 - w$ is covered by items from Step 1. If $h_3' < h_1 - 1$, then above height h_3' and until height at least $h_1 - 1$ there are everywhere items from Step 3 and therefore a width of at least $1/2 + 1/6 = 2/3$ is covered by items.

Suppose $w > 1/3$. Then only items of width in $[w, 1/2] \subset (1/3, 1/2]$ remain to be packed in Step 3(b). If $h_3' < h_1 - 1$, we have

$$\begin{aligned} \text{OPT}(L) &\geq (1-w)h_3' + \frac{2}{3}(h_1 - 1 - h_3') + 2w(h_3 - 1) && (5.4)\\ &\geq (1-w)(h_1 - 1) + 2w(h_3 - 1) && (5.5) \end{aligned}$$

If $h_3' \in (h_1 - 1, h_1]$, then we have (5.5) immediately. By combining (5.5) with (5.1), it can be seen that

$$\text{ALG}_1(L) = h_1 + h_3 \leq \frac{3}{2}\text{OPT}(L) + 2.$$

Suppose $w \leq 1/3$. Then a width of at least $2/3$ is covered in the part of the strip that is used in Step 3(b), apart from at most two parts of total height at most 2. We find

$$\text{OPT}(L) \geq \frac{2}{3}(h_1 - 1 + h_3 - 2)$$

and therefore

$$\text{ALG}_1(L) = h_1 + h_3 \leq \frac{3}{2}\text{OPT}(L) + 3.$$

□

It is straightforward to see that the complexity of our algorithm is $O(n \log n)$, where n is the number of items to be packed, since apart from sorting by width or height all the steps in the algorithm take linear time. We conjecture that the complexity of any algorithm with an approximation ratio strictly below $3/2$ is substantially higher than $O(n \log n)$.

5.1.2 An asymptotic polynomial-time approximation scheme

Since the input rectangles may be rotated, we define arbitrarily that the width is the *shorter* side of the rectangle. (It may be that the item is packed such that the width is the vertical size.) Thus for each rectangle, $0 < w_i \leq h_i \leq 1$.

The algorithm in the previous section packs only items that are wider than $1/2$ optimally.

We define

$$H_L = \sum_{i=1}^{n} h_i\,.$$

This is the height of a stack of the items in L when all these items are placed with h_i as their vertical dimension, and is an upper bound for the height of any stack of items in L.

Let $A(L)$ denote the total area of the items in L. This is independent of rotations.

We need to pack the items in L into a strip of width 1 and unbounded height. We will use the following constants in our algorithm: ε' which is the cutoff width for "small" rectangles, and $m = (1/\varepsilon')^3$ which is the number of groups used in the rounding described in the previous section. The value ε' will be determined by the desired approximation accuracy ε.

Few and wide items

We are given n items with m_1 different heights and m_2 different widths. All widths (and heights) are at least ε' and at most 1. There are at most $m_1 m_2$ groups of items, where each group contains identical items. Denote the number of items in group i by n_i ($i = 1, \ldots, m_1 m_2$). Then for group i, there are at most $n_i + 1$ possibilities with regard to the rotation. If the items in this group are squares, the orientation of the items does not matter (there is only one orientation). Otherwise, we can place k_i items such that the width is the vertical dimension and $n_i - k_i$ items such that the height is the vertical dimension, for each $k_i = 0, 1, \ldots, n_i$.

In total, this implies we have

$$\prod_{i=1}^{m_1 m_2} (n_i + 1)$$

distinct possibilities for the orientations of the items. Each such possibility is called an orientation of the input list L. It can be represented by a vector $\mathbf{v}$ with $m_1 m_2$ coordinates.

After an orientation of the list has been fixed, we can find a nearly optimal solution for packing these items using a reduction to fractional strip packing as in [107]. Since $\sum_{i=1}^{m_1 m_2} n_i = n$, the number of orientations is polynomial in the number of items: it is at most

$$\left(\frac{n}{m_1 m_2} + 1 \right)^{m_1 m_2}$$

which is polynomial in n for given (fixed) m_1 and m_2. Thus we can find the (nearly) best possible packing for these items in polynomial time by solving all the fractional strip packing problems.

Equation (2.1) implies that for a given, fixed orientation $\mathbf{v}$ of the items, this gives us a packing with height at most

$$\text{FSP}(L, \mathbf{v}) + 2m(\mathbf{v}) + 1 \leq \text{OPT}(L, \mathbf{v}) + 2m(\mathbf{v}) + 1, \tag{5.6}$$

where $\text{FSP}(L, \mathbf{v})$ and $\text{OPT}(L, \mathbf{v})$ are the optimal fractional strip packing and the optimal strip packing for the input list L using orientation $\mathbf{v}$, respectively, and $m(\mathbf{v}) \leq m_1 m_2$ is the number of distinct widths in orientation $\mathbf{v}$.

Rounding procedure

First, the input list $L_{general}$ will be partitioned into two sublists. L_{narrow} will contain all items with width at most ε' and L will contain all other items. The items in L_{narrow} will be ignored

for the time being. The dimensions of the items in L are going to be rounded up according to a procedure which we will now describe.

We are going to start by applying a geometric rounding to both heights and widths. In this way, we obtain an input instance with a limited number of distinct heights and widths, of which the total area is still close to the total area of the original input instance.

However, this is by itself not enough to ensure that the solution of the (fractional) strip packing problem for this modified problem instance is close to the real solution. Suppose we fix an orientation and the problem instance contains many items of horizontal size just smaller than $1/2$. If these widths get rounded up to a value which is just larger than $1/2$, it is clear that the height of the optimal strip packing may change by an arbitrarily high amount.

In the original paper [107], this problem was solved by transforming adjustments of the widths (horizontal sizes) into a *vertical* adjustment of the problem instance, for which it was trivial to calculate the effect on the height of the optimal strip packing. To enable us to do this in the current problem, we are going to do something similar: in fact our rounding is going to be a mixture of geometric rounding and the grouping and rounding method from [107]. We formalize this idea below.

Let I be the set $L \cup \text{rotate}(L)$. That is, for each rectangle (w_i, h_i) in L, I contains both (w_i, h_i) and (h_i, w_i). Thus $|I| = 2|L|$.

From I we can obtain a modified list I_+ such that $I \leq I_+$, as described in section 2.2.2. We use $m-1$ threshold rectangles at heights $y = ih(I)/m$ for $i = 1, \dots, m-1$.

Consider a rectangle (w_i, h_i) in L. Suppose that in I, the first corresponding rectangle (w_i, h_i) has been rounded to (w_i', h_i) while the other rectangle, (h_i, w_i), has been rounded to (h_i', w_i). We then round the original rectangle (w_i, h_i) to

$$(\min(w_i', \tilde{w}_i), \min(h_i', \tilde{h}_i)),$$

where $\tilde{x}$ is the first power of $1/(1+\varepsilon')$ greater than or equal to x.

Property 5.1 *The number of groups created in the rounding procedure is bounded by a constant.*

Proof The rounding is only applied to items that have height and width larger than ε'. Thus the highest class that can contain items is m' which is the solution of $(1+\varepsilon')^{-m'} = \varepsilon'$. We find

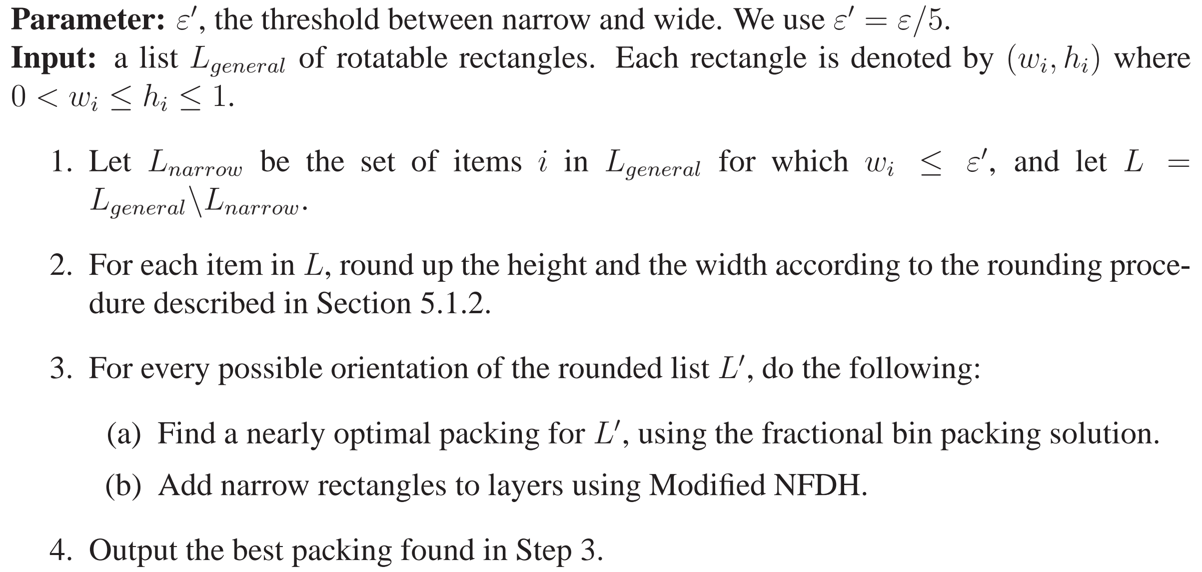
$$m' = \frac{-\log \varepsilon'}{\log(1+\varepsilon')} \approx \frac{1}{\varepsilon'} \log \frac{1}{\varepsilon'} < \frac{m}{50}$$

for $\varepsilon' < 1/12$ (recall that $m = (1/\varepsilon')^3$). The grouping and rounding allows at most m additional different widths and heights, since there are $m-1$ threshold rectangles.

This implies that the number of different widths, as well as the number of different heights, is bounded by

$$m + m' < \frac{51}{50} m$$

for $\varepsilon' < 1/12$. □


Parameter: ε', the threshold between narrow and wide. We use $\varepsilon' = \varepsilon/5$.
Input: a list $L_{general}$ of rotatable rectangles. Each rectangle is denoted by (w_i, h_i) where $0 < w_i \leq h_i \leq 1$.

1. Let L_{narrow} be the set of items i in $L_{general}$ for which $w_i \leq \varepsilon'$, and let $L = L_{general} \backslash L_{narrow}$.
2. For each item in L, round up the height and the width according to the rounding procedure described in Section 5.1.2.
3. For every possible orientation of the rounded list L', do the following:
   (a) Find a nearly optimal packing for L', using the fractional bin packing solution.
   (b) Add narrow rectangles to layers using Modified NFDH.
4. Output the best packing found in Step 3.

Figure 5.1: The APTAS summarized

Algorithm

The algorithm is summarized in Figure 5.1.

Analysis of the APTAS

In this subsection, we will prove the following theorem.

Theorem 5.3 *There is an algorithm* ALG *which, given a list* L *of* n *rectangles with side lengths at most 1 that can be rotated* $90°$, *and a positive number* $\varepsilon < 1/2$, *produces a packing of* L *in a strip of width 1 and height* $\text{ALG}(L)$ *such that*

$$\text{ALG}(L) \leq (1+\varepsilon)\text{OPT}(L) + O\left(\frac{1}{\varepsilon^3}\right).$$

The time complexity of ALG *is polynomial in n.*

The remarks in Subsection 5.1.2 immediately imply that the running time of our algorithm is polynomial in n for any fixed $\varepsilon > 0$. It is left to show the upper bound on the asymptotic performance ratio. We begin by proving two auxiliary lemmas.

Lemma 5.1 *Denote by* M *the set of rectangles* L *in optimal orientation, i.e. the orientation which allows the shortest strip packing. The list* L' *obtained after the grouping in Step 2 of the algorithm, considered in the same orientation as* M, *is such that*

$$\text{FSP}(L') \leq \left(1 + \varepsilon' + \frac{4}{(\varepsilon')^2 m}\right)\text{FSP}(M)$$

and

$$A(L') \le \left(1 + \varepsilon' + \frac{4}{(\varepsilon')^2 m}\right) A(M).$$

Proof From M we can obtain a modified list M_+, as described in Subsection 2.2.2. This time we define $m\varepsilon'/2$ threshold rectangles intersecting the lines

$$y = \frac{2H_M}{\varepsilon' m} i \qquad i = 0, \ldots, \frac{m\varepsilon'}{2}.$$

We define M_+ by increasing the width of each rectangle to the width of the *second* threshold rectangle below it. By (2.2), we have

$$\text{FSP}(M_+) \le \text{FSP}(M) + \frac{4H_M}{\varepsilon' m}. \tag{5.7}$$

However, we do not necessarily have that $L' \le M_+$, because item sizes were rounded up in both dimensions to get L'. To get a larger list, we multiply the vertical sizes of all items in M_+ by $1 + \varepsilon'$. This gives us a list M_{++}. It is clear that an equal scaling of the *vertical* size of all the items implies that the optimal fractional strip packing solution is scaled up by the same amount, i.e. by $1 + \varepsilon'$. We conclude

$$\text{FSP}(M_{++}) = (1 + \varepsilon')\text{FSP}(M_+). \tag{5.8}$$

We claim that $M_{++} \ge L'$.

Clearly, the stack for L' is not higher than the stack for M_{++}, since both dimensions of items in L' were increased by at most a factor of $(1 + \varepsilon')$. To see that the stack is also nowhere wider than the stack of M_{++}, consider the width of an item i in L'. This width is at most equal to w_j, the width of a threshold rectangle j in the stack for I (and not for L'). Note that items i and j are separated by a stack of height at most H_I/m in the I-stack.

The M-stack is a subset of the I-stack, and in the M-stack, item i is separated from its second-below threshold rectangle by a stack of height at least

$$\frac{2H_M}{\varepsilon' m}.$$

Finally, we have

$$H_M \ge \varepsilon' H_L \ge \varepsilon' H_I/2,$$

where the first inequality follows because all items have width at least ε', and the second inequality follows because in the L-stack, all items are placed with their height (largest dimension) vertically.

This implies that $2H_M/(\varepsilon' m) \ge H_I/m$. Therefore the width of the second threshold rectangle below item i in the M-stack has width at least w_j, which shows that the item i in L' does not extend further to the left than in M. This proves

$$M_{++} \ge L'. \tag{5.9}$$

We conclude that

$$\begin{aligned} \text{FSP}(L') &\le \text{FSP}(M_{++}) && \text{by (5.9)} \\ &= (1+\varepsilon')\text{FSP}(M_+) && \text{by (5.8)} \\ &\le (1+\varepsilon')\left(\text{FSP}(M) + \frac{4H_M}{\varepsilon' m}\right) && \text{by (5.7).} \end{aligned}$$

It can be seen that these statements are also valid if we replace $\text{FSP}(\cdot)$ by $A(\cdot)$ everywhere. Since all rectangles have width at least ε', we have $\varepsilon' H_M \le A(M) \le \text{FSP}(M)$. This implies the statements of the lemma. □

The following lemma follows from Lemma 4 in [107].

Lemma 5.2 *For a fixed orientation, let $L'' = L' \cup L_{narrow}$. If the height h_f at the end of Step 3(b) is larger than the height h' of the packing for L', then*

$$h_f \le \frac{A(L'')}{1-\varepsilon'} + \frac{102}{25}m + 1.$$

Proof The proof in [107] shows that in this case, nearly all the area in the packing is covered by items, apart from an additive constant which is determined by the number of different widths in the input sequence.

We now find $\frac{102}{25}m + 1$ instead of the original $4m + 1$ because by Property 1, there are at most $\frac{51}{50}m$ different widths in any orientation. By the results in [107], the number of layers in the strip packing for L' is bounded by twice the number of different widths, and at each interface between layers, there is a height of at most 2 which is not used by NFDH. □

Lemma 5.1 implies that

$$A(L'') \le A(L_{general})\left(1 + \varepsilon' + \frac{4}{(\varepsilon')^2 m}\right).$$

Therefore if $h_f > h'$, we have

$$h_f \le \frac{A(L_{general})}{1-\varepsilon'}\left(1 + \varepsilon' + \frac{4}{(\varepsilon')^2 m}\right) + \frac{102}{25}m + 1. \tag{5.10}$$

We are now ready to prove Theorem 5.3. Consider the orientation of the items in L in the optimal strip packing, i.e. the list M. At some point, our algorithm will try an orientation of L' which corresponds to this orientation. From here on, we only consider the height of the packing that our algorithm finds for this particular orientation. Clearly, the height that it outputs in Step 4 cannot be higher than this.

By (5.10), we have either $\text{ALG}(L) \le h'$ or

$$\text{ALG}(L) \le \left(1 + \varepsilon' + \frac{4}{(\varepsilon')^2 m}\right)\frac{A(L_{general})}{1-\varepsilon'} + \frac{102}{25}m + 1$$

where h' is the height of the strip packing produced in Step 3(a) of the algorithm. By (5.6) and Lemma 5.1, using as in Lemma 5.2 that the number of different widths is at most $\frac{51}{50}m$ in any orientation, we find

$$\begin{aligned} h' &\leq \text{FSP}(L') + \frac{51}{25}m + 1 \\ &\leq \left(1 + \varepsilon' + \frac{4}{(\varepsilon')^2 m}\right) \text{FSP}(M) + \frac{51}{25}m + 1 \\ &\leq \left(1 + \varepsilon' + \frac{4}{(\varepsilon')^2 m}\right) \text{OPT}(M) + \frac{51}{25}m + 1 \\ &\leq \left(1 + \varepsilon' + \frac{4}{(\varepsilon')^2 m}\right) \text{OPT}(L_{general}) + \frac{51}{25}m + 1. \end{aligned}$$

Since $A(L_{general}) \leq \text{OPT}(L_{general})$, this implies

$$\text{ALG}(L) \leq \left(1 + \varepsilon' + \frac{4}{(\varepsilon')^2 m}\right) \frac{\text{OPT}(L_{general})}{1 - \varepsilon'} + \frac{102}{25}m + 1.$$

Taking $m = (1/\varepsilon')^3$ and $\varepsilon' = \varepsilon/(6 + \varepsilon)$, this implies that

$$\text{ALG}(L_{general}) \leq (1 + \varepsilon)\text{OPT}(L_{general}) + \frac{1121}{\varepsilon^3} + 1$$

for $\varepsilon < 1/2$. This proves Theorem 5.3. We note that it is even possible to give a *fully* polynomial time approximation scheme using an LP-based approach [94].

5.2 Two-dimensional bin packing

To begin, consider again the approximation schemes from the previous section. Take $\tilde{\varepsilon} = \varepsilon/2$. Suppose our algorithm with as input parameter $\tilde{\varepsilon}$ gives a strip packing with a height of H for a given set of items. Denote the height of the optimal *strip* packing by H'. We have

$$H \leq (1 + \tilde{\varepsilon})H' + O\left(\frac{1}{\tilde{\varepsilon}^2}\right).$$

Given this strip packing, we can define cuts at all integer heights. Then starting from the cut at height 1, we can allocate the items to square bins as follows: put all the items below this cut into a separate bin (the upper edge of some of these items may coincide with the cut), and put all the items that intersect the cut into another bin. Continue with the next higher cut until all items are packed into bins.

Using the above method, we put the items into at most $2H$ bins. Additionally, there does not exist a packing into bins which uses less than H' bins, because such a packing could trivially be turned into a strip packing that uses a height at most $H' - 1$, which is a contradiction.

Thus the number of bins that we use to pack the items is at most $2 + 2\tilde{\varepsilon} = 2 + \varepsilon$ times the optimal number of bins required for these items, plus an additive constant.

In the current section, we define an alternative approximation algorithm which is slightly worse. However, it has the advantage that it can easily be adapted to the more complex problems listed below. Moreover, it is much faster.

Items that have width and height greater than $1/2$ are called *big*. The following definitions will be used throughout this chapter and the next.

Definition 5.1 *An item is in* standard orientation *if its height is at least as large as its width, otherwise it is in* reverse orientation.

All items are placed in standard orientation. Since we are going to use this algorithm as a subroutine for three-dimensional problems, we denote the dimensions of the items by widths and *lengths* (instead of heights) from now on to avoid confusion later. We begin by partitioning the items into types. We use the following intervals.

- $(2/3, 1]$ (type 0)
- $(1/2, 2/3]$ (type 1)
- $(1/(i+1), 1/i]$ for $i = 2, \ldots, 8$ (type i)
- $(0, 1/9]$ (type 9)

A two-dimensional item is of type (i, j) if its width is of type i and its length is of type j. Clearly $i \geq j$ due to the orientation we defined. In some cases, we will use a finer classification for type 1. We let type $1a = (1/2, 11/20]$, type $1b = (11/20, 3/5]$ and type $1c = (3/5, 2/3]$.

Large types There are four types which we will call *large*. We will begin by defining their packing. Each such type is packed independently of the other ones. We pack items of types $(1, 1)$, $(1, 0)$ and $(0, 0)$ one per bin, always in the left bottom corner of the bin and in standard orientation. The items of type $(2, 1)$ are further classified according to their length (largest dimension). Items of type $(2, 1a)$ are packed two per bin, both in reverse orientation, touching the same edge of the bin and each other, with one of them in the left bottom corner of the bin. The same holds for items of type $(2, 1b)$ and $(2, 1c)$ (but the items of these three subtypes are not packed together in any bin).

Medium types There are also four *medium* types. Items of type $(2, 2)$ are packed four per bin (the bin is first partitioned into four identical sub-bins). Items of type $(i, 0)$ are packed i per bin for $i = 2, 3, 4$.

After the packing of the large and medium types is completed, smaller items are added. They are first added into bins which contain large items. If some items remain unpacked after those bins are considered, they are packed into empty bins.

Bins containing items of the types $(0, 0)$, $(2, 2)$, $(2, 0)$, $(3, 0)$ and $(4, 0)$ do not receive smaller items. We note that all these bins are packed so that a fraction of at least $4/9$ of their area is occupied, except possibly the last bin for each of the last four types. This follows from the types and the amounts of items per bin.

Approximation ratio The performance bound of $9/4$ follows from one of the two following reasons.

1. If no new bins are opened for smaller items, we use a weighting function for the analysis. Those functions are usually useful in analyzing on-line algorithms. Here we use it to analyze an offline algorithm.

2. If at least one bin was opened for smaller items, we use an area based analysis. We show that all bins except a constant number have items of total area of at least $4/9$. Then we get $OPT \geq (4/9)(ALG_2 - c)$ which implies the performance ratio.

Case 1 The weighting function is defined in the following way. Small items get weight 0.

Type	$(0,0)$	$(1,0)$	$(1,1)$	$(2,1)$	$(2,0)$	$(2,2)$	$(3,0)$	$(4,0)$
Weight	1	1	1	$1/2$	$1/2$	$1/4$	$1/3$	$1/4$

The following claim is immediate from the definitions, and explains our choice of weighting function.

Claim 5.1 *All bins packed by our algorithm with large items, except possibly the last one for each (sub)type, contain a weight of 1.*

Claim 5.2 *A bin can contain at most nine items of both width and length larger than* $1/4$.

Proof See Chapter 4, Claim 4.4. □

It follows from the same result that a bin can contain at most twenty-five items of both width and length larger than $1/6$.

Claim 5.3 *A bin can contain at most 9/4 of weight.*

Proof Consider a bin with a certain amount of weight. We may assume there is no item of type $(0,0)$ or $(1,0)$, because the smaller type $(1,1)$ also has weight 1, and also no item of type $(2,0)$ because $(2,1)$ gives the same weight.

We will use Claim 5.2 to determine the highest possible weight in a bin by expressing all items as multiples of items of width and length just larger than $1/4$ or $1/6$. For instance, by cutting a (1,1) item halfway both horizontally and vertically, it can be seen that other items of 'worth' at most 5 items of width and length just larger than $1/4$ can be placed with it in one bin (otherwise this cutting would create a packing with more than 9 such items, contradicting Claim 5.2).

An overview can be found in the following table. Here the heading ‘items $> 1/4$’ means ‘number of items of length and width more than $1/4$ that items of this type contain’, etc.

Type	items $> 1/4$	items $> 1/6$	weight	weight per item $> 1/6$
$(1,1)$	4	9	1	$1/9$
$(2,1)$	2	6	$1/2$	$1/12$
$(2,2)$	1	4	$1/4$	$1/16$
$(3,0)$	2	4	$1/3$	$1/12$
$(4,0)$	0	4	$1/4$	$1/16$

If there is no item of type $(1,1)$, then by the last column, the weight per ‘virtual’ item of width and length larger than $1/6$ is at most $1/12$ which gives total weight of at most $25/12 < 9/4$.

Otherwise, by the second column and Claim 5.2, at most 2 items of type $(2,1)$ or $(3,0)$ can be in the bin together with the item of type $(1,1)$. To get maximum weight, we should maximize the number of virtual items that have weight $1/12$ per item. We can have at most 12 such virtual items because there can be at most 2 items that cover 6 of them. This leaves at most 4 virtual items with weight per item $1/16$, giving additional weight of $1/4$. The total weight therefore is at most 1 (from the largest item) $+1$ (from the $(2,1)$ items) $+1/4 = 9/4$. $\square$

Case 2 It is left to show how small items are packed to keep a $4/9$ fraction of each bin occupied (except for a constant number of bins). Each bin will contain items of a given small type or set of types. For each type or set of types, we need to show how they are packed in the following three cases.

A. A bin that already contains a $(1,0)$ item, or two $(2,1)$ items.

B. A bin that already contains a $(1,1)$ item.

C. An empty bin.

Consider the area left for further packing in the three cases. See Figure 5.2. For many small types, summarized in Table 5.1, the packing of the small items does not depend on the exact size of the large items that they are packed with.

In type A bins, there is a strip of width $1/3$ and length 1 that does not contain any items. Such a bin already contains an area of at least $1/3$.

In type B bins, the area outside of a square of $2/3$ by $2/3$ in the left bottom corner does not contain any items. We partition this L-shaped area in two rectangles, one of dimensions 1 and $1/3$ and the other of dimensions $2/3$ and $1/3$. The orientation is not important since rotations are allowed. We pack some number of small items in the larger rectangle and some number in the smaller rectangle; the numbers are written as a sum in the ‘items’ column for type B. These bins already contain an area of at least $1/4$.

In type C bins, we use the so-called side-by-side packing [58] to pack items. I.e., for type (i,j) items, we place ij of these items in an i by j grid at the bottom of the bin, and then (when possible) add some extra items in reverse orientation at the top of the bin.

Type		A		B		C	
		items	area $= 1/3+$	items	area $= 1/4+$	items	area
$(3,1)$		1	$1/8$	$1+1$	$1/4$	4	$1/2$
$(3,2)$		2	$1/6$	$2+1$	$1/4$	6	$1/2$
$(4,2)$		2	$2/15$	$2+1$	$1/5$	8	$8/15$
(i,j)	$3 \le j \le i \le 4$	3	$3/25$	$3+2$	$1/5$	ij	$9/16$
$(5,j)$	$j = 3,4,5$	5	$5/36$	$5+3$	$2/9$	$5j$	$5/8$
$(i,1)$	$i = 6,7,8$	2	$1/9$	$2+2$	$2/9$	8	$4/9$
$(i,2)$	$i = 6,7,8$	4	$4/27$	$4+2$	$2/9$	12	$12/27$
$(i,3)$	$i = 6,7,8$	6	$1/6$	$6+4$	$5/18$	18	$1/2$
(i,j)	$i = 6,7,8$ $j = 4,5$	8	$4/27$	$8+4$	$2/9$	24	$4/9$
(i,j)	$6 \le j \le i \le 8$	12	$12/81$	$12+8$	$20/81$	36	$4/9$

Type		shelves	area $= 1/3+$	shelves	area $= 1/4+$	shelves	area
$(9,i)$	$i = 2,\ldots,8$	i	$2/3 \cdot 2/9$	$i+1$	$2/9$	i	$8/9 \cdot 2/3$

Table 5.1: All types that are combined on a single line of the table are packed together, except the types (i,j) for $3 \le j \le i \le 5$ in empty bins (type C bins) and the $(9,i)$ types.
For the $(9,i)$ types, shelves of length $1/i$ and width $1/3$ are created in type A and B bins. They are all filled to a length of $1/(i+1)$ and a width of $2/9$. In bins of type B, one extra shelf is created in the smaller part of the L-shape. In bins of type C, shelves of width 1 are created; they are filled to a width of $8/9$.

Type	Bins	Condition	items	area
$(4,1)$	A	subtype $(4,1)_a$	1	$1/3+9/80$
		$w>11/20$	1	$11/30+1/10$
		$w\leq 11/20$	2	$1/3+2/10$
	B		$1+1$	$1/4+1/5$
	C		5	$1/2$
$(5,0)$	A		1	$1/3+1/9$
	B	$w>3/5$	$1+0$	$9/25+1/9$
		$w\leq 3/5$	$2+0$	$1/4+2/9$
	C		5	$5/9$
$(5,1)$	A	$w>3/5$	1	$2/5+1/12$
		$w\leq 3/5$	2	$1/3+1/6$
	B	$w>3/5$	$1+1$	$9/25+1/6$
		$w\leq 3/5$	$2+1$	$1/4+1/4$
	C		6	$1/2$
$(5,2)$	A		2	$1/3+1/9$
	B	$w>3/5$	$2+1$	$9/25+3/18$
		$w\leq 3/5$	$4+1$	$1/4+5/18$
	C		10	$5/9$
$(i,0), i=6,7,8$	A		2	$1/3+4/(3i+3)$
	B	subtype $(i,0)_a$	$2+1$	$1/4+2/(i+1)$
		subtype $(i,0)_b$	$2+0$	$1/4+2\frac{i-1}{i}\frac{1}{i+1}$
	C		i	$2/3\cdot i/(i+1)$

Type	Bins	Condition	shelves	area
$(9,i), i=0,1$	A		1	$1/3+2/9\cdot 1/2$
$(9,0)$	B	$w>11/20$	1	$\frac{121}{400}+\frac{2}{3}\frac{2}{9}$
		$w\leq 11/20$	1	$1/4+(\frac{9}{20}-\frac{1}{9})$
$(9,1)$	B		2	$1/4+4/9\cdot 1/2$
$(9,i), i=0,1$	C		1	$1/2\cdot 8/9$

Table 5.2: The variable w in the Condition column refers to the width of the big item(s) in the current bin of type A or B. Recall that the width is the *smallest* size of an item.
The type $(4,1)_a$ contains items of width in the interval $(9/40,1/4]$. The type $(i,0)_a$ $(i=6,7,8)$ contains items of width in $(2/3,\frac{i-1}{i}]$, the type $(i,0)_b$ contains items of width in $(\frac{i-1}{i},1]$. The types $(i,0)$ $(i=4,\ldots,8)$ are packed separately (in type B bins: both subtypes separately), the types $(9,0)$ and $(9,1)$ are packed together in type A and C bins. For the $(9,1)$ items in type B bins, we use two shelves of length 1 and $2/3$, both of width $1/3$. In both shelves, at least a width of $2/9$ and length of $1/2$ will be occupied.

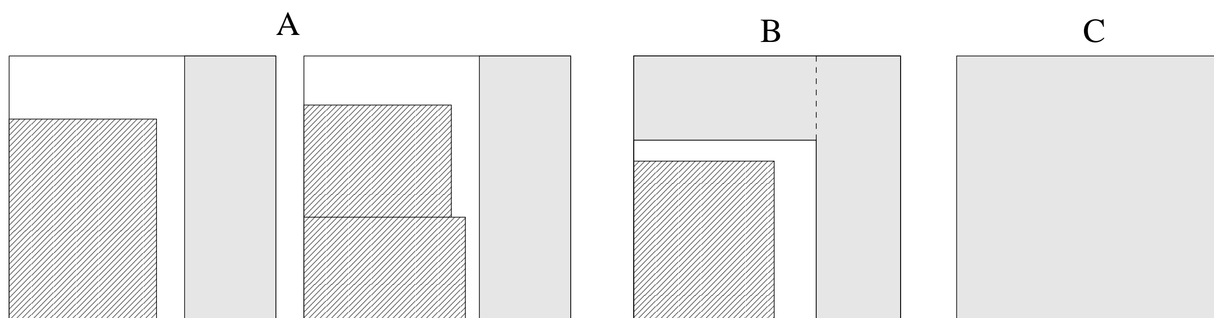


Figure 5.2: Unused areas in bins of types A, B and C. Small items are packed here

Table 5.2 contains the items that are slightly more complicated to pack, at least in Type A and Type B bins. Here it is usually important what the exact width of the large items is. This is the reason that we classified type (2,1) items further: we can now be assured that if one of them has e.g. width $w \leq 3/5$, the other one has this as well. (Note: to keep the analysis uniform, for these items we let the width be the *largest* size. The width of a pair can be taken arbitrarily as either the width of the first or the second item, due to our classification.)

We explain the column marked 'Condition'. The entries marked 'subtype' are explained in the caption. An entry $w \leq x$ or $w > x$ refers to the width of the large item(s) in this bin; in the cases marked $w \leq x$, we use vertical strips of width $1 - x$ to pack small items, instead of width $1/3$. The horizontal strips (in type B bins) always have width $1/3$. For items of type $(6, 0)$, $(7, 0)$ and $(8, 0)$, the width of the vertical strip in type B bins depends on the width of the *small* items packed in there.

The area that is already in a type A or B bin is of course different if we put restrictions on the width of the large items; it is given by the first term of the sum in the 'area' column. Finally, there is one type that still remains to be packed. This type, $(9, 9)$, is described below.

Type $(9, 9)$ We show how to pack these items into square sub-bins of width and length $1/3$ so that inside each such sub-bin at least $4/9$ of its area is occupied. We begin by showing that this is a dense enough packing in all three cases.

A. We can create three sub-bins. We get a total area of $1/3 + 12/81 = 13/27 > 4/9$.

B. The item already packed in this bin has length and width no larger than $2/3$. Therefore we can create five sub-bins. The total occupied area would be $1/4 + 5 \cdot 4/81 = 161/324 > 4/9$.

C. We create nine sub-bins and get a total area of $4/9$.

Next we explain the packing into sub-bins. We use Next Fit Decreasing Length (NFDL) to pack items into these sub-bins. All items are put in standard orientation (length $\geq$ width), and then sorted by decreasing length. Then, the items are packed into levels using Next Fit, where the length of each level is the length of the first item placed in it. When the next item does not fit in the current level anymore, a new level is started, if necessary in a new sub-bin.

Since all items have width at most $1/9$, we find that each level is filled to a width of at least 2/9. Denote the length of level i by H_i. Let k be the number of levels in the current bin. The first item that does not fit has length H_{k+1}. All items in level i have length at least H_{i+1}.

Then for each sub-bin except the last, the packed area is at least

$$\frac{2}{9}(H_2 + ... + H_{k+1}) > 2/9(1/3 - H_1) > 4/81.$$

This is a $4/9$ fraction of the area of the sub-bin which is $1/9$.

Theorem 5.4 *For any input list L, we have*

$$\text{ALG}_2(L) \leq 9/4 \cdot \text{OPT}(L) + 41.$$

Proof If no new bins are opened for small items, we have from Claim 5.1 that there are at most 7 bins with weight less than 1 (this cannot occur for the types $(0,0), (1,0)$ and $(1,1)$). Combining this with Claim 5.3 gives

$$\text{ALG}_2(L) \leq 9/4 \cdot \text{OPT}(L) + 7.$$

If there are new bins opened for small items, then almost all bins contain an area of at least $4/9$. By the above analysis, this holds in this case for all bins that contain small items, except possibly the last such bin for each type that is packed separately. Note that it is also possible that we run out of a certain small type while we are packing a bin of type A, in this case this bin is not used further and has a bad area guarantee. Counting the number of types packed separately, there are 21 such types in Table 5.1 and 12 in Table 5.2. (Note that the type $(4,1)$ can only cause a single bin with a low area guarantee, because this cannot happen for subtype $(4,1)_a$ or for a bin of type A with $w > 11/20$.) Finally there is the type $(9,9)$.

Moreover, there can be at most 7 bins with large items that have no small items and area less than $4/9$: these are the bins that had weight less than 1 after packing the large items. Since the total area of the items is a lower bound on the optimal number of bins required to pack them, we find in this case

$$\text{ALG}_2(L) \leq 9/4 \cdot \text{OPT}(L) + 41.$$

□

5.3 This side up

We now show how to use the algorithm in the previous section to get a $9/4$-approximation algorithm for the This Side Up problem. Naively, one might think that one could simply group items of similar height and pack each group using the algorithm from the previous section (ignoring the height of the items). However, the problem with this is that some groups might contain only large items and other groups only small items. In this case, the groups with large items will have poor volume guarantees, and we will not get a good approximation ratio.

We therefore have to be more careful. Our algorithm works as follows. All items are classified into types as in the previous section, where the height of the items is (for now) ignored. We then begin by packing the large items. The $(0,0)$ items are stacked in some way, nothing is placed next to this stack.

Subtypes For the $(1,0)$, $(1,1)$ and $(2,1)$ items, we classify them further along a dimension that has type 1, using the types $1a$, $1b$ and $1c$ that were defined at the start of section 5.2. Thus we have in total nine subtypes (the $(1,1)$ items are only classified further along their width (smallest size)). We sort the items by subtype, items of subtypes of $(2,1)$ are further sorted by decreasing height.

For each subtype, the items are stacked in the strip so that one corner of each item is directly above a designated corner of the base of the strip, and all items are oriented in the same way. Pairs of $(2,1)$ items are considered as a single $(1,0)$ item in this step, where the width is one of the lengths (largest sizes) of the pair. Thus we have six stacks of items on top of each other: three for the $(1,1)$ items and one each for items of types $(1x,0)$ and $(2,1x)$ for $x=a,b,c$. Here we rotate the $(2,1)$ items such that their length is oriented along the width of the $(1,0)$ items, and a free strip is left next to these items along one side of the main strip.

Packing small items If we view any one of these stacks from above, it leaves either an L-shaped area or one strip. We can now start using this extra volume for the small items, using the six stacks one by one. The small items are also packed per type. Within each type, the items are sorted in order of decreasing height. Then the items are packed in layers, where each layer is packed as in section 5.2 next to the current stack. This is not directly possible for the $(9,9)$-items, we discuss these separately below. The height of a layer is the height of the first item packed in it.

Because this stack contains items of only one large subtype (by considering pairs of $(2,1)$ items as $(1,0)$ items), and the layers contain items from one small type, the packing uses the same unique method on all layers that are used for this small type. It is for this reason that we can ignore the single large items in the stack and only care about the height of the stack. If we did not use this distinction into subtypes for the large items, we might need to change the packing method many times, and we would leave much vertical space unused.

We continue creating layers until we run out of items for this small type or the next small item does not fit next to the current stack (its height would be higher than the height of the stack). In this last case, the remaining items of this type are packed next to the next stack of large items. I.e., the next layer for this type is not created immediately above the previous layer, but instead at the height where the next stack starts. Also, the packing method might be changed at this point.

Finally, if all six stacks are used in this way, or the small items are all packed, we pack the remaining medium and/or small items according to the methods for packing items into empty bins. That is, for each (medium or small) type, the items are sorted according to height and then packed in order of decreasing height using the methods from section 5.2, using as many layers as necessary.

The $(9,9)$-items These items are sorted by decreasing length in section 5.2, but now we need to sort them by height. We therefore pack them as follows. Consider the sequence of unpacked $(9,9)$-items, sorted by decreasing height. We assume the items are numbered $1,2,\ldots,n_1$. Using binary search, we find the largest i such that the items $1,\ldots,i$ can be packed into a sub-bin. To find a packing for a given set of items (i.e. a given value of i) in a sub-bin, we use NFDL. In this

way, each item packed in a given sub-bin is no lower than each item in the next sub-bin with $(9, 9)$-items.

Moreover, since we use NFDL within a sub-bin, we still have the area guarantee of $4/9$. To prove this, consider an index i such that the items $1, \dots, i$ can be packed into a sub-bin, whereas the items $1, \dots, i, i + 1$ can not. Consider the packing which would be created by NFDL on the items $1, \dots, i, i + 1$. Since these items do not actually fit into the sub-bin, the last non-empty shelf is overloaded, i.e., the total width of items in it is strictly larger than 1. We show that removing the item of index $i + 1$ from the packing still leaves a packed area of at least $4/9$. Note that the packing of NFDL for these items may be different and we are just proving the area guarantee for the items that can clearly be packed using NFDL.

We use the indices $1, 2, \dots, s$ for the shelves, where the shelf s is empty or of height zero. Let j be the shelf on which the item of index $i + 1$ is packed. If the removal of this item leaves the shelf with total width of at least $2/3$, we stop. Otherwise, assume first that $j < s - 1$. Take the leftmost item z from shelf $j + 1$, and move it to shelf j. Note that this item fits into the shelf since its height is at least the height of the moved item z. Moreover, the total width of the previous contents of shelf j (before removing an item from there) together with z is larger than 1 (otherwise, z would have been packed on shelf j). Therefore the total width without the removed item is at least $2/3$ (since the width of z is at most $1/3$). The total width with z but without the removed item is at most 1, since the width after the removal is less than $2/3$. We let $j = j + 1$ (since now shelf $j + 1$ is lacking one removed item) and repeat this until either the process is stopped, or $j = s - 1$. Since shelf $s - 1$ is overloaded, a removal of one item does not decrease its total width below $2/3$ (note that it may be still overloaded after the removal). We get that all shelves have a total width of at least $2/3$, and the contents of each shelf $t < s$ are at least of the height of shelf $t + 1$ (since we only moved the highest rectangle of each shelf one shelf down). Therefore, also a height of $2/3$ is covered, which implies an occupied area of at least $4/9$.

Theorem 5.5 *For any input list L, we have*

$$\text{ALG}_3(L) \leq 9/4 \cdot \text{OPT}(L) + 45.$$

Proof We begin by making a general remark. Whenever items are packed into layers in order of decreasing height, some height in each layer is lost because the first item on the layer determines the height of the layer, and the next items might have smaller height. However, if we denote the heights of the layers by $H_1, \dots, H_k$, we have that all items on layer i have height at least H_{i+1}. To see how much area is occupied, we can move all items from each layer i to layer $i + 1$. Then layer $i+1$ is completely covered by items for each i, and only layer 1 is left empty. This implies that when we consider the height of the entire packing for these items, at most a height of H_1 does not contain any items. We have two cases in our analysis.

Case 1 All the small items fit next to the six created stacks. In this case, we can ignore the small items in the analysis because they do not add to the total height used. In this case, we can use the weighting technique from section 5.2.

The weight of an item is now defined as the vertical size divided by the number of times that the 'horizontal item' fits in a square. To give a bound on the performance ratio, we introduce a new concept which is the *weight density* of an item. This is the weight of an item *per unit of vertical dimension*, i.e. it is the weight of the two-dimensional item that we get when we ignore the vertical dimension of an item. The weight density of an arbitrary horizontal plane through a packing of items is the sum of the weight densities of the items which intersect with this plane. We will examine the weight densities at arbitrary horizontal planes through the packings of our algorithm and of the optimal packing.

We find that for each large (sub)type, if it is packed in layers between heights h_1 and h_2, the weight density is 1 at all heights $h \in [h_1, h_2]$ apart from a total height of at most 1. For $(0, 0)$ and the subtypes of $(1, 1)$ and $(1, 0)$, there is even a weight density of 1 at the entire height of their stacks, because all these items are placed directly on top of each other. In the optimal solution, according to Claim 5.3, there can not be a weight density of more than $9/4$ at any height. Since we use seven types that do not have a weight of 1 everywhere (four medium types and the three subtypes of $(2, 1)$), we find that

$$\text{ALG}_3(L) \leq 9/4 \cdot \text{OPT}(L) + 7.$$

Case 2 Some small items need to be packed above the large items. Consider some large (sub)type (one stack) and a single small type t. Suppose all items from this type are placed next to this large subtype, between heights h_t and h'_t. Since the small items are sorted by decreasing height, and the large items are all stacked on top of each other, we have for each height $h_t \leq h \leq h'_t$ an area guarantee of $4/9$ using the proof from section 5.2, apart from a total height of at most 1.

A small type may also be split among two large stacks, or among one stack and layers of its own (not next to any stack). In this case, some height at the top of the first stack might not contain small items. We can assign this additional height loss to the large (sub)type of that stack. We then find that for each large and small (sub)type, there is a height of at most 1 at which we do not have an area guarantee of $4/9$. In total we have 10 large (sub)types in separate stacks and $21 + 13 + 1 = 35$ small (sub)types and we find

$$\text{ALG}_3(L) \leq 9/4 \cdot \text{OPT}(L) + 45.$$

□

5.4 Further applications

5.4.1 Three-dimensional strip packing

To pack items in a three-dimensional strip, we place each "large" item such that its weight, defined as in the previous section, is minimized. Note that this does not mean simply placing it with its smallest dimension vertical, because the number of times that the implied horizontal item fits in a square might depend on the orientation.

Extending the standard orientation from before, let w, ℓ, and h be the smallest, second smallest and third smallest dimension of an item. We describe in the table below how the items which we will call large are placed.

Type	Condition	Vertical dimension	Weight	Type (2d)
$(0,0,0),(1,0,0)$		w	w	$(0,0)$
$(1,1,0)^*$		w	w	$(1,0)^*$
$(1,1,1)^*$		w	w	$(1,1)^*$
$(2,0,0)$	$\ell \leq 2w$	ℓ	$\ell/2$	$(2,0)$
	$\ell > 2w$	w	w	$(0,0)$
$(2,1,0)$		ℓ	$\ell/2$	$(2,0)$
$(2,1,1)^*$		ℓ	$\ell/2$	$(2,1)^*$
$(2,2,0)$	$h \leq 2w$	h	$h/4$	$(2,2)$
	$h > 2w$	w	$w/2$	$(2,0)$
$(2,2,1)$		h	$h/4$	$(2,2)$
$(2,2,2)$		w	$w/4$	$(2,2)$
$(3,0,0)$	$\ell \leq 3w$	ℓ	$\ell/3$	$(3,0)$
	$\ell > 3w$	w	w	$(0,0)$
$(4,0,0)$	$\ell \leq 4w$	ℓ	$\ell/4$	$(4,0)$
	$\ell > 4w$	w	w	$(0,0)$

Table 5.3: How to pack large items in a three-dimensional strip

As an example, consider the items of type $(2,0,0)$. For these items we have $w \in (1/3,1/2]$ and $\ell > 2/3$. The most efficient packing for these items depends on the ratio $\ell : w$. If this ratio is at least 2, the items are relatively long and it is best to place them with their smallest dimension (w) vertical. However, if the ratio is less than 2, we can place two items side by side and need only a height of ℓ to place two such items, whereas with w as vertical dimension we would need $2w > \ell$ since then we cannot place the items next to eachother. Thus within each type in the table, all items are placed as efficiently as possible.

The last column contains the type of the item when the vertical dimension is ignored. For each line of the table, all items are stacked in layers in order of decreasing height. Naturally items that map to the same two-dimensional type can be stacked together in a single stack (e.g. items in lines 1, 5, 13 and 15). The height of a layer is the height of the first item placed in that layer.

All types not mentioned in this table are placed with their *largest* dimension, h, vertically, leaving a small two-dimensional type. These items are called *small*. Small items are combined with some large items (the ones marked with * in the table) exactly as in the This Side Up algorithm from the previous section. That is, for each a and b such that (a,b) is a small type, all types of the form (a,b,x) are combined into a single type. Items are sorted in order of decreasing height and packed into layers next to existing stacks, or at unused heights.

Theorem 5.6 *For any input list L, we have*

$$\text{ALG}_3(L) \leq 9/4 \cdot \text{OPT}(L) + 45.$$

Proof As before, we have two cases. Suppose first that all small items can be placed next to the large items. It can be seen that our algorithm has a weight density of 1 at all heights apart from a total height of at most 7, as before. For the optimal packing, we use an unusual definition of the weights and let the weight of any large item be the vertical size of this item *as packed in the optimal packing*, divided by the number of times that the 'horizontal item' fits in a square. Denote the sum of the weights of the large items in input list L when packed by algorithm $\mathcal{A}$ as $W_{\mathcal{A}}(L)$. Since our algorithm places the large items such that their weight is minimized, we have for the total weights that $W_{\text{ALG}_4}(L) \leq W_{\text{OPT}}(L)$.

Moreover, it is still the case that the weight density at any horizontal plane through the optimal packing is at most $9/4$, since this is still equal to the total weight of the corresponding two-dimensional items packed in the square at that height. Thus we find $\text{OPT}(L) \geq 4/9 \cdot W_{\text{OPT}}(L)$.

We have

$$\text{ALG}_4(L) \leq W_{\text{ALG}_4}(L) + 7 \leq W_{\text{OPT}}(L) + 7 \leq 9/4 \cdot \text{OPT}(L) + 7.$$

Now suppose that some small items are placed above all large items. In this case, we find as in the analysis of the This Side Up problem that at all heights apart from a total height of at most 45, an area of at least $4/9$ is occupied by items. Since the optimal algorithm must pack the entire volume, the performance ratio of $9/4$ again follows: $\text{ALG}_4(L) \leq 9/4 \cdot \text{OPT}(L) + 45$. □

5.4.2 Three-dimensional bin packing

Finally, we can turn the packing generated by ALG_4 into a packing for the fully rotatable three-dimensional bin packing problem. This is done by making horizontal cuts at all integer heights. Then, for each cut starting from the bottom we do the following:

- all items below this cut and above the next lower cut are packed into an empty bin
- all items that are intersected by this cut are packed into an empty bin

Finally, all items above the last cut are packed into an empty bin. Since all items have height at most 1, it is clear that this generates a valid packing. Moreover, if we place the bins on top of each other in a stack, it is clear that the height of the highest bin is at most twice the height achieved by ALG_4 plus 1, and the number of bins required to pack L can not be smaller than the optimal height to pack L in a strip.

We conclude

$$\text{ALG}_5(L) \leq 9/2 \cdot \text{OPT}(L) + 91.$$

Although almost all our algorithms are based on ALG_2, this is the only direct reduction where no further modifications to the algorithm are required. More formally, any algorithm for three-dimensional strip packing with rotations that has performance ratio $\mathcal{R}$ can be turned into an algorithm for three-dimensional bin packing with rotations that has performance ratio $2\mathcal{R}$.

The above algorithm and analysis can be also applied to the "This side up" problem in bins.

5.5 Conclusion

In this paper we design offline algorithms for six packing problems. Most of these problems were not studied in on-line environments, which can be interesting as well. It might be the case that some of the bounds in this chapter can be improved. Specifically we are interested in improving the constant for packing in three-dimensional bins (both for rotatable items and for the "This Side Up" problem). This can be done by designing algorithms for these problems directly instead of adaptation of algorithms for other problems.

Part II

Scheduling

Chapter 6

Minimizing the total completion time online on a single machine, using restarts

We examine the scheduling problem of minimizing the total completion time (the sum of completion times) online on a single machine, using restarts. Each job has a size which is the amount of time it needs to be run in order to complete. Allowing restarts means that the processing of a job may be interrupted, losing all the work done on it. In this case, the job must be started again later (restarted), until it is completed without interruptions. We study the online problem, where algorithms must decide how to schedule the existing jobs without any knowledge about the future arrivals of jobs.

We compare the performance of an online algorithm $\mathcal{A}$ to that of an optimal off-line algorithm $\textsc{opt}$ that knows the entire job sequence σ in advance. The total completion time of an input σ given to an algorithm $\textsc{alg}$ is denoted by $\textsc{alg}(\sigma)$. The competitive ratio $\mathcal{R}(\mathcal{A})$ of an online algorithm $\mathcal{A}$ is defined as

$$\mathcal{R}(\mathcal{A}) = \sup_{\sigma} \frac{\mathcal{A}(\sigma)}{\textsc{opt}(\sigma)} .$$

Known results For the case where all jobs are available at time 0, the shortest processing time algorithm $\textsc{spt}$ [148] has an optimal total completion time. This algorithm runs the jobs in order of increasing size. Hoogeveen and Vestjens [90] showed that if jobs arrive over time and restarts are not allowed, the optimal competitive ratio is 2, and they gave an algorithm $\textsc{dspt}$ ('delayed $\textsc{spt}$') which maintained that competitive ratio.

We are aware of three previous instances where restarts were proven to help. First, in [145] it was shown that restarts help to minimize the makespan (the maximum completion time) of jobs with unknown sizes on m related machines. Here each machine has its own speed, which does not depend on the job it is running. The algorithm in [145] obtains a competitive ratio of $O(\log m)$. Without restarts, the lower bound is $\Omega(\sqrt{m})$.

Second, [1] shows that restarts help to minimize the maximum delivery time on a single machine, obtaining an (optimal) competitive ratio of $3/2$ while without restarts, $(\sqrt{5}+1)/2$ is the best possible. In this problem, each job needs to be delivered after completing, which takes a certain given extra time.

Third, in [89] it is shown that restarts help to minimize the number of *early* jobs (jobs that complete on or before their due date) on a single machine, obtaining an (optimal) competitive ratio of 2 while without restarts, it is not possible to be competitive at all (not even with preemptions).

Our results Until now, it was not known how to use restarts in a deterministic algorithm for minimizing the total completion time on a single machine to get a competitive ratio below 2, whereas a ratio of 2 can be achieved by an algorithm that does not restart. We give an algorithm RSPT ('restarting SPT') of competitive ratio 3/2. This ratio cannot be obtained without restarts, even with the use of randomization.

Our algorithm is arguably the simplest possible algorithm for this problem that uses restarts: it bases the decision about whether or not it will interrupt a running job J for an arriving job J' solely on J and J'. It ignores, for example, all other jobs that are waiting to be run. We show in section 6.1 that the analysis of our algorithm is tight and that all "RSPT-like" algorithms have a competitive ratio of at least 1.48. This suggests that a more complicated algorithm would be required to get a substantially better competitive ratio, if possible.

6.1 Algorithm RSPT

We present our online algorithm RSPT for the problem of minimizing the total completion time on a single machine, using restarts. See Figure 6.1. We define a job J to be *running at time* t if it started at time $s < t$, it did not complete before time t and no other job started to run in (s, t).

Figure 6.1: The algorithm RSPT

RSPT maintains a queue Q of unfinished jobs. A job is put into Q when it arrives. A job is removed from Q when it is completed. For any time t, RSPT deals first with all arrivals of jobs at time t before starting or interrupting any job.
At any time t where either RSPT completes a job, or one or more jobs arrive while RSPT is idle, RSPT starts to run the smallest (remaining) job in Q. If $Q = \emptyset$, RSPT is idle (until the next job arrives).
Furthermore, if at time r a job J is running that started (most recently) at time $s < r$ and has size x, and if at time r a new job J' arrives with size w, then RSPT interrupts J and starts to run J' if and only if

$$r + w \leq \frac{2}{3}(s + x). \tag{6.1}$$

Otherwise, RSPT continues to run J (and J' is put into Q).

RSPT has the following properties (where J, x, s, r and w are defined as in Figure 6.1). OPT is any optimal off-line algorithm (there can be more than one).

R1 RSPT only interrupts a job J for jobs that are smaller and that can finish earlier than J.
Proof. If there is an interruption, $w \leq \frac{2}{3}(s + x) - r < \frac{2}{3}x - \frac{1}{3}s \leq x$ and $r + w < s + x$.

R2 If RSPT does not interrupt J for a job of size w that arrives at time r, then $r+w > \frac{2}{3}(s+x)$. Furthermore, if RSPT is still running J at time $r+w$, it runs J until completion.
Proof. The first claim holds by condition (6.1) in the algorithm. To see the other claim, observe that any job J' that arrives after time $r+w$ satisfies $r'+w' \geq r' \geq r+w > \frac{2}{3}(s+x)$ in this case, and does not cause an interruption. □

R3 Suppose that $s \leq t \leq \frac{2}{3}(s+x)$, and RSPT has been running J continuously from time s until time t. Then at time t, all jobs smaller than J that are completed by OPT are also completed by RSPT.
Proof. The property holds for $t = s$ by definition of RSPT. For $t > s$, a smaller job that OPT completed and RSPT did not, can thus only have arrived after time s. But then it would have caused an interruption of J before time t, since it can be completed before time $\frac{2}{3}(s+x)$. □

R4 Suppose that $s < t \leq \frac{2}{3}(s+x)$, and RSPT has been running J continuously from time s until time t. Then at time t, OPT has completed at most one job that RSPT has not completed.
Proof. By R3, the only jobs that OPT can have already completed at time t that RSPT has not, have size at least x. However, we have $t < 2x$, since $t \geq 2x \Rightarrow \frac{2}{3}(s+x) \geq 2x \Rightarrow s \geq 2x \Rightarrow \frac{1}{3}s \geq \frac{2}{3}x \Rightarrow s \geq \frac{2}{3}(s+x) \Rightarrow t > \frac{2}{3}(s+x)$, a contradiction. Before time $t < 2x$, OPT can complete at most one job of size at least x and therefore at most one job that RSPT has not yet completed. □

R5 At any time t, RSPT only interrupts jobs that it cannot finish before time $\frac{3}{2}t$. Hence, RSPT does not interrupt any job with a size of at most half its starting time.
Proof. If there is an interruption at time t, then a job arrived at time t for which $t+w \leq \frac{2}{3}(s+x)$, hence without interruptions J would have finished at time $s+x \geq \frac{3}{2}(t+w) \geq \frac{3}{2}t$. □

To explain some of the intuition behind the definition of RSPT, we define a family of similar algorithms and prove that none of them can do much better than RSPT. We define a family of algorithms, $\{\text{RSPT}(\alpha)\}$, $\alpha \in [0,1]$ as follows: RSPT(α) behaves exactly like RSPT, but (6.1) is replaced by

$$r + w \leq \alpha(s+x).$$

It is possible that RSPT(α) outperforms RSPT for some value of α. However, we show that the improvement could only be very small, if any. Therefore, to keep the analysis manageable, we analyze only RSPT.

Lemma 6.1 *For all* $0 < \alpha < 1$, $\mathcal{R}(\text{RSPT}(\alpha)) \geq 1.48$.

Proof. We consider three job sequences. We use a small constant $\varepsilon > 0$.

1) A job of size 1 arrives at time 0, and N jobs of size 0 arrive at time $\alpha+\varepsilon$. RSPT will run these jobs in order of arrival time and have a total completion time of $N+1$. However, it is possible to obtain a total completion time of $(\alpha+\varepsilon)(N+1)+1$ by running the jobs of size 0 first. By letting N grow without bound, the competitive ratio tends to $1/\alpha$ for $\varepsilon \to 0$.

Sequence 1) shows that $\mathcal{R}(\text{RSPT}(\alpha)) \geq 3/2$ for $\alpha \leq 2/3$. For the remainder of this proof, assume that $\alpha > 2/3$.

2) A job J_1 of size 1 arrives at time 0, a job J_2 of size α at time ε, a job J_3 of size 0 at time α (causing an interruption in $\text{RSPT}(\alpha)$). Then, at time $\alpha + \varepsilon$ one final job J_4 of size $\alpha(2\alpha) - \alpha$ arrives. For this sequence, the optimal cost tends to $4\alpha^2 + 2\alpha + 1$ (using the job sequence J_2, J_3, J_4, J_1) whereas $\text{RSPT}(\alpha)$ completes the jobs in the order J_3, J_2, J_4, J_1 and pays $4\alpha^2 + 5\alpha + 1$.

3) As 2), but after job J_4, at time $2\alpha^2$ another job J_5 of size 0 arrives, causing $\text{RSPT}(\alpha)$ to interrupt job J_2 which it is running at that time. Starting at time $2\alpha^2$, $\text{RSPT}(\alpha)$ runs the jobs J_5, J_4, J_2 and J_1 in this order. The optimal order of the jobs is J_2, J_3, J_4, J_5, J_1. In the limit, $\text{RSPT}(\alpha)$ pays $14\alpha^2 + 1$ whereas the optimal cost is $6\alpha^2 + 2\alpha + 1$.

This implies that

$$\mathcal{R}(\text{RSPT}(\alpha)) \geq \max\left(\frac{1}{\alpha}, \frac{4\alpha^2 + 5\alpha + 1}{4\alpha^2 + 2\alpha + 1}, \frac{14\alpha^2 + 1}{6\alpha^2 + 2\alpha + 1}\right) > 1.48008.$$

□

From the first job sequence in the proof, we have the following corollary.

Corollary 6.1 $\mathcal{R}(\text{RSPT}) \geq 3/2$.

We will return to sequences similar to 2) and 3) in Section 6.7.

To analyze the competitive ratio of RSPT, we will use amortized analysis [22]. Each job that arrives receives a certain amount of credit, based on its (estimated) completion time in the optimal schedule and in RSPT's schedule. We will show that each time that RSPT starts a job, we can distribute the credits of the jobs so that a certain invariant holds, using an induction. The calculations of the credits at such a time, and in particular of the estimates of the completion times in the two schedules, will be made under the assumption that no more jobs arrive later.

We first need to show that the invariant holds at the first time that RSPT starts a job. Then, we need to show that at each later job start, the invariant keeps holding when we take into account the jobs that arrived in the meantime (updating calculations where necessary!) and assume no more jobs arrive. Finally, we need to show that if the invariant holds at the last time that RSPT starts a job, then RSPT maintains a competitive ratio of $3/2$.

There will be one special case where the invariant does not hold again immediately. In that case, we will show the invariant is restored at some later time before the completion of σ. This case will be analyzed in Section 6.7.

6.2 Global assumptions and event assumptions

Definition 6.1 *An* event *is the start of a job by* RSPT.

Definition 6.2 *An event has the property STATIC if no more jobs arrive after this event.*

At the time of an event, RSPT completes a job, interrupts a job, or is idle.

In our analysis, we will use 'Global assumptions' and 'Event assumptions'. We show that we can restrict our analysis to certain types of input sequences and schedules and formulate these restrictions as Global assumptions. Then, when analyzing an event (from the remaining set of input sequences), we show in several cases that it is sufficient to consider events with certain properties, and make the corresponding Event assumption. The most important one was already mentioned in Section 6.1:

Event assumption 1 *The current event has the property STATIC.*

There can be more than one optimal schedule for a given input σ. For the analysis, we fix some optimal schedule and denote the algorithm that makes that schedule by OPT. We use this schedule in the analysis of every event. Hence, OPT takes into account jobs that have not arrived yet in making its schedule, but OPT does not change its schedule between successive events: the schedule is completely determined at time 0. OPT does not interrupt jobs, because it can simply keep the machine idle instead of starting a certain job and interrupting it later, without affecting the total completion time. We can make the following assumption about RSPT and OPT, because the cost of OPT and RSPT for a sequence is unaffected by changing the order of jobs of the same size in their schedules.

Global assumption 1 *If two or more jobs in σ have the same size,* RSPT *and* OPT *complete them in the same order.*

Definition 6.3 *An input sequence σ has property* SMALL *if, whenever* RSPT *is running a job of some size x from σ, only jobs strictly smaller than x arrive.*

Lemma 6.2 *For every input sequence σ, it is possible to modify the arrival times of some jobs such that the resulting sequence σ' has the property* SMALL*, the schedule of* RSPT *for σ' is the same as it is for σ, and* $\text{OPT}(\sigma') \leq \text{OPT}(\sigma)$.

Proof. At any time r that a job J arrives that is at least as large as the job that RSPT is running at that time, we modify σ as follows. If there has been an interval before time r in which RSPT was idle, define u as the end of the last such interval before r; otherwise set $u = 0$. Define r' as the last time in the interval (u, r) that a job larger than J was interrupted or completed. If there is no such time, set $r' = u$. We change the release time of J to r'.

When RSPT is run on the resulting sequence σ', it does not consider running J during the interval $[r', r]$: it is running smaller or equal-sized jobs in that entire interval. (For the equal-sized jobs, see Global assumption 1.) Hence the schedule of RSPT for σ' is the same as it is for σ, and $\text{OPT}(\sigma') \leq \text{OPT}(\sigma)$ since the optimal cost can only decrease if the arrival times decrease or remain the same. □

This Lemma implies that if RSPT maintains a competitive ratio of 3/2 on all the sequences that have property *SMALL*, it maintains that competitive ratio overall. Henceforth, we make the following assumption.

Global assumption 2 *The input sequence σ has property SMALL.*

By this assumption, if RSPT is running a job of size x, the first job of size $y \geq x$ can only arrive once RSPT starts a job of some size $z > y$, or when RSPT becomes idle.

6.3 Definitions and notation

After these preliminaries, we are ready to state our main definitions. A job J arrives at its release time $r(J)$ and has processing time denoted by $w(J)$. For a job J_i, we will usually abbreviate $r(J_i)$ as r_i and $w(J_i)$ as w_i, and use analogous notation for jobs J', J^* etc. When RSPT is running a job J, it can have both *J-large* unfinished jobs, that are at least as large as J, and *J-small* unfinished jobs, that are smaller, in its queue. To distinguish between these two sets of jobs, the unfinished J-large jobs will be denoted by $J^1 = J, J^2, J^3, \ldots$ with sizes $x_1 = x = w(J), x_2 = w(J_2), x_3 = w(J_3), \ldots$ while the J-small jobs will be denoted by $J_1, J_2, \ldots$ with sizes $w_1, w_2, \ldots$

We let $Q(t)$ denote the queue Q of RSPT at time t.

Definition 6.4 *A* run-interval *is a half-open interval $I = (s(I), t(I)]$, where* RSPT *starts to run a job (denoted by $J(I)$) at time $s(I)$ and runs it continuously until exactly time $t(I)$. At time $t(I)$, $J(I)$ is either completed or interrupted. We denote the size of $J(I)$ by $x(I)$.*

The input sequence σ may contain jobs of size 0. Such jobs are completed instantly when they start and do not have a run-interval associated with them. Thus we can divide the entire execution of RSPT into run-intervals, completions of 0-sized jobs, and intervals where RSPT is idle. The following lemma follows immediately from the definition of RSPT.

Lemma 6.3 *All jobs in σ arrive either in a run-interval or at the end of an interval in which* RSPT *is idle.*

Definition 6.5 *For any run-interval I, we denote the set of jobs that arrive during this interval by ARRIVE$(I) = \{J_1(I), \ldots, J_{k(I)}(I)\}$, We write $r_i(I) = r(J_i(I))$ and $w_i(I) = w(J_i(I))$ for $1 \leq i \leq k(I)$. The jobs are ordered such that $w_1(I) \leq w_2(I) \leq \ldots \leq w_{k(I)}(I)$. We denote the total size of jobs in ARRIVE(I) by $W(I)$, and write $W_i(I) = \sum_{j=1}^{i} w_i(I)$ for $1 \leq i \leq k(I)$.*

RSPT will run the jobs in *ARRIVE*(I) in the order $J_1(I), \ldots, J_{k(I)}(I)$ (using Global assumption 1 if necessary) and we have $w_{k(I)}(I) < x(I)$ using Global assumption 2. Of course it is possible that *ARRIVE*$(I) = \emptyset$. In that case I ends with the completion of the job RSPT was running, and we have $t(I) = s(I) + x(I)$.

Lemma 6.4 *If* RSPT *interrupts job $J(I)$, then $t(I) = r_1(I)$.*

Proof. We have $t(I) \in \{r_1(I), \ldots, r_{k(I)}(I)\}$. Note that $t(I) < r_1(I)$ is not possible, since all jobs in *ARRIVE*(I) arrive on or before time $t(I)$. Suppose $t(I) = r_i(I) > r_1(I)$ for some $i > 1$, then $r_i(I) + w_i(I) \leq \frac{2}{3}(s(I) + x(I))$. By the ordering of the jobs in *ARRIVE*(I) we have $w_i(I) \geq w_1(I)$ and thus $r_1(I) + w_1(I) < r_i(I) + w_i(I) \leq \frac{2}{3}(s(I) + x(I))$. But then RSPT interrupts $J(I)$ no later than at time $r_1(I)$, so $t(I) \leq r_1(I)$, a contradiction. $\square$

Definition 6.6 *For the jobs in ARRIVE*(I)*, we write* $\tau_i(I) = r_i(I) + w_i(I) - \frac{2}{3}(s(I) + x(I))$ $(i = 1, \ldots, k(I))$.

We have $\tau_i(I) > 0$ for $i = 2, \ldots, k(I)$, and $\tau_1(I) > 0$ if $J(I)$ completes at time $t(I)$, $\tau_1(I) \leq 0$ if it is interrupted at time $t(I)$.

Definition 6.7 *We define* $f_{\text{OPT}}(I)$ *as the index of the job that* OPT *completes first among the jobs from ARRIVE*(I).

Definition 6.8 *An interruption by* RSPT *at time* t *is* slow *if* OPT *starts to run a job which is in the set ARRIVE*(I) *strictly before time* t*; in this case,* $f_{\text{OPT}}(I) > 1$ *and* $J_{f_{\text{OPT}}(I)}(I)$ *did not cause an interruption when it arrived. Otherwise the interruption is* fast.

We call such an interruption slow, because in this case it could have been better for the total completion time of RSPT if it had interrupted $J(I)$ for one of the earlier jobs in *ARRIVE*(I) (i. e. faster); now, at time t, RSPT still has to run all the jobs in *ARRIVE*(I), whereas OPT has already partially completed $J_{f_{\text{OPT}}(I)}(I)$. Note that whether an interruption is slow or fast depends entirely on when OPT runs the jobs in *ARRIVE*(I). It has nothing to do with RSPT.

We now define some variables that can change over time. We will need their values at time t when we are analyzing an event at time t. They represent a snapshot of the current situation.

Definition 6.9 *If job* J *has arrived but is not completed at time* t*,* $s_t(J)$ *is the (next) time at which* RSPT *will start* J*, based on the jobs that have arrived until time* t*. For a job* J *that is completed at time* t*,* $s_t(J)$ *is the last time at which* J *was started (i.e. the time when it was started and not interrupted anymore). For a job* J *that has not arrived yet at time* t*,* $s_t(J)$ *is undefined.*

Lemma 6.5 *For every event and every job* J*,* $s_t(J)$ *is at least as high as it was during the previous event.*

Proof. Consider an event at time t and a job J. If J completes before or at time t, then $s_t(J)$ is unchanged since the previous event. Any other job J at time t, for which $s_t(J)$ was already defined during the previous event, is larger than the jobs in *ARRIVE*(I) by definition of RSPT and by Global assumption 2. Therefore J will complete after the jobs in *ARRIVE*(I), i.e. no earlier than previously calculated. □

By this Lemma, for a job J in $Q(t)$, $s_t(J)$ is the earliest possible time that RSPT will start to run J.

Definition 6.10 *A job* J *is* interruptible *at time* t*, if* $s_t(J) < 2w(J)$ *and* $t \leq \frac{2}{3}(s_t(J) + w(J))$.

I. e. a job J is interruptible if it is still possible that RSPT will interrupt J after time t (cf. Property R5).

Definition 6.11 $BEFORE_t(J)$ *is the set of jobs that* RSPT *completes before* $s_t(J)$ *(based on the jobs that have arrived at or before time* t*).* $b_t(J)$ *is the total size of jobs in* $BEFORE_t(J)$*.* $\ell_t(J)$ *is the size of the largest job in* $BEFORE_t(J)$*.*

Clearly, $b_t(J)$ and $\ell_t(J)$ can only increase over time, and $\ell_t(J) \leq b_t(J)$ for all times t and jobs J.

During our analysis, we will maintain an *estimate* on the starting time of each job J in the schedule of OPT, denoted by $s_t^{\mathrm{OPT}}(J)$. We describe later how we make and update these estimates. We will maintain the following inequality as part of our invariant, which will be defined in section 6.4.2. Denote the actual optimal completion time of a job J by $\mathrm{OPT}(J)$. Then at the time t of an event,

$$\boxed{\sum_{J:r(J)\leq t} \mathrm{OPT}(J) \geq \sum_{J:r(J)\leq t} (s_t^{\mathrm{OPT}}(J) + w(J))} \tag{6.2}$$

This equation implies that at the end of the sequence, $\mathrm{OPT}(\sigma) \geq \sum_J (s_t^{\mathrm{OPT}}(J) + w(J))$. We will use the following lemma to calculate initial values of $s_t^{\mathrm{OPT}}(J)$ for arriving jobs in such a way that (6.2) holds.

Lemma 6.6 *For a given time t, denote the most recent arrival time of a job by $t' \leq t$. Denote the job that* OPT *is running at time t' by $\Phi(t')$, and its remaining unprocessed jobs by $\Psi(t')$. The total completion time of* OPT *of the jobs in $\Psi(t')$ is at least the total completion time of these jobs in the schedule where those jobs are run consecutively in order of increasing size after $\Phi(t')$ is completed.*

Proof. The schedule described in the lemma is optimal in case no more jobs arrive after time t (Event assumption 1). If other jobs do arrive after time t, it is possible that another order for the jobs in $\Psi(t')$ is better overall. However, since this order is suboptimal for $\Psi(t')$, we must have that the total completion time of the jobs in $\Psi(t')$ is then not smaller. □

The fact that the optimal schedule is not known during the analysis of an event is also the reason that we check that (6.2) is satisfied instead of trying to maintain $\mathrm{OPT}(J) \geq s_t^{\mathrm{OPT}}(J) + w(J)$ for each job J separately.

Definition 6.12 *$D_t(J) = s_t(J) - s_t^{\mathrm{OPT}}(J)$ is the* delay *of job J at time t.*

6.4 Amortized analysis

The credit of job J at time t is denoted by $K_t(J)$. A job will be assigned an initial credit at the first event on or after its arrival. At the end of each run-interval $I = (s, t]$, each job $J_i(I)$ in *ARRIVE*(I) receives an initial credit of

$$\frac{1}{2}\left(s^{\mathrm{OPT}}(J_i(I)) + w_i(I)\right) - D(J_i(I)) \qquad i = 1, \ldots, k(I). \tag{6.3}$$

If at time t a (non-zero) interval ends in which RSPT is idle, or $t = 0$, then suppose $Q(t) = \{J_1, \ldots, J_k\}$ where $w_1 \leq \ldots \leq w_k$. The initial credit of job J_i in $Q(t)$ is then

$$\frac{1}{2}t + \frac{1}{2}\sum_{j=1}^{i} w(J_j) \qquad i = 1, \ldots, k(I). \tag{6.4}$$

This is a special case of (6.3): by Lemma 6.6 and Event assumption 1, OPT will run the jobs in $Q(t)$ in order of increasing size, hence $s_t^{\text{OPT}}(J_i) \geq s_t(J_i)$ for $i = 1, \ldots, k$. Therefore $D(J_i) \leq 0$ for $i = 1, \ldots, k$. Moreover, by definition of RSPT we have $s_t(J_i) = t + \sum_{j=1}^{i-1} w_i$ for $i = 1, \ldots, k$.

The idea is that the credit of a job indicates how much its execution can still be postponed by RSPT without violating the competitive ratio of $3/2$: if a job has δ credit, it can be postponed by δ time.

For the competitive ratio, it does not matter how much credit each individual job has, and we will often transfer credits between jobs as an aid in the analysis. During the analysis of events, apart from transferring credits between jobs, we will also use the following rules.

Rule C1. If $s_t(J)$ increased by δ since the previous event, then $K(J)$ decreases by δ.

Rule C2. If the estimate $s_t^{\text{OPT}}(J)$ increased by δ since the previous event, then $K(J)$ increases by $\frac{3}{2}\delta$.

$s_t(J)$ cannot decrease by Lemma 6.5. We will only adjust (increase) $s_t^{\text{OPT}}(J)$ in a few special cases, where we can show that (6.2) still holds if we increase $s_t^{\text{OPT}}(J)$. Both rules follow directly from (6.3): it can be seen that if $s_t(J)$ or $s_t^{\text{OPT}}(J)$ increases, J should have received a different amount of credit initially. (The amount that J can be postponed changes.)

Theorem 6.1 *Suppose that after* RSPT *completes any input sequence σ, the total amount of credit in the jobs is nonnegative, and (6.2) holds. Then* RSPT *maintains a competitive ratio of* $3/2$.

Proof. We can ignore credit transfers between jobs, since they do not affect the total amount of credit. Then each job has at the end credit of

$$K(J) = \frac{1}{2}(s_t^{\text{OPT}}(J) + w(J)) - (s_t(J) - s_t^{\text{OPT}}(J)),$$

where we use the final (highest) value of $s_t^{\text{OPT}}(J)$ for each job J, and the actual starting time $s_t(J)$ of each job in RSPT's schedule. This follows from (6.3) and the rules for adjusting job credits mentioned above. Thus if the total credit is nonnegative, we have

$$\begin{aligned} \sum_J (s_t(J) - s_t^{\text{OPT}}(J)) &\leq \frac{1}{2}\sum_J \left(s_t^{\text{OPT}}(J) + w(J)\right) \\ \Rightarrow \sum_J s_t(J) &\leq \frac{3}{2}\sum_J s_t^{\text{OPT}}(J) + \frac{1}{2}\sum_J w(J) \\ \Rightarrow \text{RSPT}(\sigma) = \sum_J (s_t(J) + w(J)) &\leq \frac{3}{2}\sum_J (s_t^{\text{OPT}}(J) + w(J)) \leq \frac{3}{2}\text{OPT}(\sigma). \qquad \square \end{aligned}$$

Calculating the initial credit The only unknowns in (6.3) are $s_t^{\text{OPT}}(J_i(I))$ $(i = 1, \ldots, k(I))$. If there is an interruption at time t, Lemma 6.6, together with the job that OPT is running at time t, gives us a schedule for OPT that we can use to calculate valid estimates (lower bounds) $s_t^{\text{OPT}}(J_i(I))$ for all i. We also use the following Event assumption.

Event assumption 2 *If the run-interval I ends in a completion, all jobs in ARRIVE(I) arrive no later than the time at which* OPT *completes $J_{f_{\mathrm{OPT}}(I)}(I)$.*

We briefly explain this assumption. By definition, all jobs in *ARRIVE*(I) arrive no later than at time $t(I)$. By the time OPT completes $J_{f_{\mathrm{OPT}}(I)}(I)$, RSPT will not interrupt $J(I)$ anymore by Property R2. Whether other jobs in *ARRIVE*(I) arrive at that time or at some later time $\leq t(I)$ does not affect RSPT's decisions or its total completion time. Event assumption 2 enables us to apply Lemma 6.6 to calculate lower bounds for the completion times of OPT of the jobs in *ARRIVE*(I). Some release times might actually be higher, but the optimal total cost for σ cannot be lower in that case. Therefore (6.2) holds.

Note that if we were to modify the sequence σ by actually decreasing release times until Event assumption 2 holds (similarly to in Lemma 6.2), the optimal schedule for the resulting sequence might be quite different. In particular, $f_{\mathrm{OPT}}(I)$ may change! This is the reason we use this assumption only locally, to get some valid lower bounds on the optimal cost.

Note also that both after a completion and after an interruption, the schedule of OPT is not completely known even with these assumptions, because we do not know which job OPT was running at time t. Therefore we still need to consider several off-line schedules in the following analysis.

6.4.1 Credit requirements

In this section, we describe three situations in which credit is required, and try to clarify some of the intuition behind the invariant defined in Section 6.4.2.

Interruptions Suppose a job J of size x is interrupted at time r_1, because job J_1 arrives, after starting at time s. Then $s < 2x$. J_1 will give away credit to J, J^2, J^3 and J^4 as described in Table 6.1, and nothing to any other jobs. We briefly describe the intuition behind this. We have the following properties.

INT1 The amount of lost processing time due to this interruption is $r_1 - s$. This is at most $\frac{2}{3}(s+x) - s = \frac{2}{3}x - \frac{s}{3}$, which is monotonically decreasing in s.

INT2 The size of J_1 is w_1. This is at most $\frac{2}{3}(s+x) - r_1 < \frac{2}{3}(s+x) - s = \frac{2}{3}x - \frac{s}{3}$, which is monotonically decreasing in s.

So, in Table 6.1, J_1 appears to give away more credit if s is larger, but a) it has more (this follows from (6.3); b) it needs less (we will explain this later); and c) $r_1 - s$ is smaller.

From Table 6.1 we can also see how much credit is still missing. For instance if $s < x$ and $x < r_1$, then J^2 receives $r_1 - x$ from J_1, but it lost $r_1 - s$ because it now starts $r_1 - s$ time later. We will therefore require that in such a case, J^2 has at least $x - s$ of credit itself, so that it still has nonnegative credit after this interruption. In general, any job that does not get all of its lost credit back according to the table above, must have the remaining credit itself. We will formalize this definition in Section 6.4.2.

J_1 will only give credit to jobs that actually exist (at this time). By Global assumption 2, a J-large job which does not yet exist can only arrive during the execution of an even larger job,

s	$[0,x]$	$[0,x]$	$(x,\frac{3}{2}x]$	$(x,\frac{3}{2}x]$	$(\frac{3}{2}x,2x)$
r_1	$[0,x]$	$(x,\frac{4}{3}x]$	$(x,\frac{3}{2}x]$	$(\frac{3}{2}x,\frac{5}{3}x]$	$(\frac{3}{2}x,2x)$
To J	r_1-s	r_1+w_1-s	r_1+w_1-s	r_1+w_1-s	r_1+w_1-s
To J^2	0	r_1-x	r_1-s	r_1-s	r_1-s
To J^3	0	0	0	$r_1-\frac{3}{2}x$	r_1-s
To J^4	0	0	0	$r_1-\frac{3}{2}x$	r_1-s
Total	r_1-s	$2r_1+w_1$ $-(s+x)$	$2(r_1-s)+w_1$	$4r_1+w_1$ $-2s-3x$	$4(r_1-s)+w_1$

Table 6.1: Credit given by J_1 to other jobs

or when RSPT becomes idle. In both of these cases, we will calculate the initial credit for such a job at the point at which it arrives, based on its arrival time, and we will not use the credit that it might have received from this job J_1. Thus any credit that according to this table would be given to a non-existing job (a job that has not arrived yet, or will not arrive) is simply lost.

Completions Suppose a job J completes at time $s+x$. We give the following property without proof.

COM1 The jobs in *ARRIVE*(I) (where $I=(s,s+x]$) need to get at most $\frac{1}{2}(x-b_s(J))$ of credit from J.

By this ("needing" credit) we mean that the amount of credit those jobs receive initially, together with at most $\frac{1}{2}(x-b_s(J))$, is sufficient for these jobs to satisfy the conditions that we will specify in the next section.

Small jobs As long as a job J has not been completed yet, it is possible that smaller jobs than J arrive that are completed before J by RSPT. If OPT completes them after J, then $D_t(J)$ increases.

6.4.2 The invariant

From the previous section we see that for a job, sometimes credit is required to pay for interruptions of jobs that are run before it, (e.g. on page 92 below, J^2 pays $x-s$ for an interruption of J), and sometimes to make sure that jobs that arrive during its final run have sufficient credit (COM1). We will make sure that each job has enough credit to pay both for interruptions of jobs before it and for its own completion (i.e. for jobs that arrive during its final run).

For a job J, we define the *interrupt-delay* associated with an interruption as the amount of increase of $D_t(J)$ compared to the previous event. This amount is at most $t-s$ at the end of a run-interval $(s,t]$. (It is less for a job J if $s_{\mathrm{OPT}}(J)$ also increases).

Credit can also be required because the situation marked "Small jobs" in Section 6.4.1 occurs. The *small job-delay* of J associated with an event at time t is the total size of jobs smaller than J in *ARRIVE*(I) that are completed before J by RSPT and after J by OPT.

From Table 6.1, we can derive bounds for the amount of credit that J^2 and later jobs should have themselves so that they still have nonnegative credit after this interruption. We denote this

amount by $N_{INT}(J_i, t)$ at time t (*N*eeded credit for *INT*erruptions).

Consider an event at time t. Suppose $Q(t) = \{J_1, \dots, J_k\}$, where $w_1 \le w_2 \le \dots \le w_k$. We write $s_i = s_t(J_i) = t + \sum_{j=1}^{i-1} w_j$. From Table 6.1 it can be seen that

$$N_{INT}(J_i, t) = \max(0, w_{i-1} - s_{i-1}) + \max(0, \frac{3}{2}w_{i-2} - s_{i-2}) + \max(0, 2w_{i-4} - t), \quad (6.5)$$

where each maximum only appears if the corresponding job exists. For the third maximum in this equation, note that the total interrupt-delay of J_i caused by interruptions of the jobs $J_1, \dots, J_{i-4}$ is at most $2w_{i-4} - t$ after time t, since RSPT starts to run J_1 at time t and does not interrupt any of the jobs $J_1, \dots, J_{i-4}$ after time $2w_{i-4}$ by Property 4. Using a simple case analysis, we can see that in all cases

$$N_{INT}(J_i, t) \le \max(0, w_{i-1} + \frac{1}{2}w_{i-2} + \frac{1}{2}w_{i-4} - t). \quad (6.6)$$

For any existing job $J \notin Q(t)$, i.e. it has already been completed, we define $N_{INT}(J, t) = 0$.

When a job J completes, it will transfer its credit to the jobs that arrived during the last run interval in which it was running. In Section 6.6, we will describe in detail how this credit is distributed. Job J will never give away more than $\frac{1}{2}(w(J) - b_t(J))$ if it completes at time t. We therefore define $N_{COM}(J_i, t)$ (*N*eeded credit for *COM*pletions) as follows: for a job $J_i \in Q(t)$, we have

$$N_{COM}(J_i, t) = \max\left(0, \frac{1}{2}(w_i - b_t(J_i))\right). \quad (6.7)$$

For any job J that is already completed at time t, we define $N_{COM}(J, t) = 0$.

For all jobs J that have arrived at time t, we will maintain

$$\boxed{K_t(J) \ge N_{COM}(J, t) + N_{INT}(J, t).} \quad (6.8)$$

Invariant We now define our invariant, that will hold at specific times t in the execution, and in particular when a sequence is completed:

Invariant: At time t, for all jobs that have arrived, (6.8) holds; furthermore, (6.2) holds.

Theorem 6.2 $\mathcal{R}(\text{RSPT}) = 3/2$.

Proof outline. The proof consists of a case analysis, which makes up the rest of this paper. In the rest of this section we show that (6.2) can be maintained and that (6.8) holds for large jobs. In section 6.5 and beyond, we consider all possible interruptions and completions. “All possible” refers to both the times at which these events occur, and the possible schedules of the off-line algorithm.

For every possible event, we will give a time at which the above invariant holds again, assuming that it held after the previous event. This will be no later than at the completion of the last job in σ. At that time, the invariant implies that all completed jobs have nonnegative credit,

since for completed jobs we have $N_{COM}(J,t) = N_{INT}(J,t) = 0$. Also, (6.2) holds. We can then apply Theorem 6.1.

We divide the analysis into the following cases.

1. An interruption of a job J (Lemmas 6.12, 6.13, 6.14) in all but one case, Case 3 below
2. Completion of a job J (Lemmas 6.15, 6.16, 6.17, 6.18)
3. A slow interruption of a job J of size x in the case that RSPT started it before time $2x/3$ (Section 6.7), and OPT does not run any J-large jobs before *ARRIVE*(I).

For almost all events, it will be the case that the invariant holds again immediately after the current event. However, there is one event for which it takes slightly longer: this is a slow interruption of a job J, where $s(J) \leq \frac{2}{3}w(J)$. If such an event occurs at some time t, we will show that the invariant is restored no later than when RSPT has completed the second-smallest job in *ARRIVE*(I). (This job exists by Lemma 6.4 and Definition 6.8.)

In order to ensure that the invariant holds again after an event, we will often transfer credits between jobs. Also, we will use the credit that some jobs must have because the invariant was true previously, to pay for their interrupt-delay or for their completion. We need to take into account that $N_{INT}(J,t)$, $s_t^{\mathrm{OPT}}(J)$ etc. of some jobs that arrived before or at the previous event can change as a result of the arrival of new jobs, compared to the calculations in that event (that were made under the assumption that STATIC held). By the discussion following Theorem 6.1, (6.2) holds at each event if it held at the previous event and if $s_t^{\mathrm{OPT}}(J)$ is not changed for any job J that arrived at or before the previous event. We also have the following lemma.

Lemma 6.7 *Suppose $Q(t) = \{J_1, \ldots, J_k\}$, where $w_1 \leq w_2 \leq \ldots \leq w_k$, and* RSPT *starts J_1 at time t. A job J_i satisfies (6.8) in any of the following situations.*

1. $K_t(J_i) \geq \max(0, \frac{1}{2}\sum_{j=1}^{i} w_j - t)$.
2. $K_t(J_i) \geq \frac{1}{2}(w_i - w_{i-1}) + \frac{1}{2}\sum_{j=1}^{i-2} w_j$ *and* $t \geq w_{i-1}$
3. $K_t(J_i) \geq \frac{1}{2}(w_i - w_{i-1}) + \frac{1}{2}\sum_{j=1}^{i-2} w_j + (w_{i-1} - t)$ *and* $t < w_{i-1}$
4. $K_{t'}(J_i) \geq \frac{1}{2}(w_i - w_{i-1})$ *and J_i starts to run at time t'*

Proof. Note first of all that $w_{i-1} \leq \ell_t(J_i)$ because RSPT runs the jobs in order of increasing size.

1. We have $\frac{1}{2}\sum_{j=1}^{i} w_j - t = \frac{1}{2}(w_i - w_{i-1}) + w_{i-1} + \frac{1}{2}\sum_{j=1}^{i-2} w_j - t \geq \frac{1}{2}(w_i - \ell_t(J)) + N_{INT}(J_i,t) \geq N_{COM}(J_i,t) + N_{INT}(J_i,t)$.

2. Here we have $N_{INT}(J_i,t) \leq \frac{1}{2}(w_{i-2} + w_{i-4})$, since $t \geq w_{i-1} \geq w_{i-2} \geq w_{i-4}$. Hence $K_t(J_i) \geq \frac{1}{2}(w_i - w_{i-1}) + \frac{1}{2}\sum_{j=1}^{i-2} w_j \geq N_{COM}(J_i,t) + N_{INT}(J_i,t)$.

3. Now $N_{INT}(J_i,t) \leq \frac{1}{2}(w_{i-2} + w_{i-4}) + (w_{i-1} - t)$ and we are done similarly.

4. This can only happen if $J_1, \ldots, J_{i-1}$ are completed, because RSPT runs jobs in order of size. We have $N_{INT}(J_i,t') = 0$ by (6.6), since $t' \geq \sum_{J':J' \text{ completed}} w(J')$. Furthermore, since $w_{i-1} \leq \ell_{t'}(J_i)$, $K_{t'}(J_i) \geq \frac{1}{2}(w_i - \ell_{t'}(J_i)) \geq N_{COM}(J_i,t)$. □

Notation	Definition	Long notation
t	time of the current event	
s	start of the most recent run-interval $I = (s, t]$	
J	job that RSPT was running in $I = (s, t]$	$J(I)$
x	its size	$w(J(I))$
ARRIVE	jobs that arrive in I	*ARRIVE*(I)
k	number of jobs in *ARRIVE*	$k(I)$
W	total size of jobs in *ARRIVE*	$W_{k(I)}(I)$
J_1	smallest job that arrives in I	$J_1(I)$
r_1	its arrival time	$r_1(I)$
w_1	its size	$w_1(I)$
f	index of job in *ARRIVE* that OPT runs first	$f_{\text{OPT}}(I)$

Table 6.2: Notations

Corollary 6.2 *Suppose* RSPT *starts to run jobs at time t, where $t = 0$ or t is the end of a (nonzero) interval in which* RSPT *was idle. Then (6.8) and (6.2) hold for the jobs that arrive at time t.*

Proof. This follows directly from (6.4) and Lemma 6.7, Case 1. □
We can apply this corollary at the arrival time of the first job in σ, and anytime after RSPT has been idle.

6.4.3 Analysis of an event

As described in the previous subsection, for the analysis of RSPT we need to analyze every possible event that can occur during its execution, i. e. show that the invariant holds after the event, if it holds after the previous event. For each event, we will focus on the credits of jobs at the time of the current event, denoted by t. Generally, we will drop the subscript t and write $K(J_i)$ for each job J_i. Furthermore, the job that was interrupted or completed at the time of the event will be denoted by J, and the set of smaller jobs that arrived during the most recent run of J will be denoted by *ARRIVE* $= \{J_1, \dots, J_k\}$. The most recent starting time of J will be denoted by s. We will call J-large jobs *large*, and others *small*. Remember that f_{OPT}, abbreviated by f, is the index of the job in *ARRIVE* that OPT completes first. Our notation is summarized in Table 6.2.

Lemma 6.8 *After* OPT *completes J_f, then if STATIC holds,* OPT *does not run any J-large job until all jobs in ARRIVE are completed.*

Proof. This is a direct consequence of Lemma 6.6 and Event assumptions 1 and 2: after J_f, OPT will complete first the remaining jobs in *ARRIVE* in order of increasing size, and then the remaining large jobs - as long as no new jobs arrive. □

Using this lemma, the schedule of OPT is completely determined by which large jobs it runs before *ARRIVE* (the rest will be run after, in order of increasing size).

Lemma 6.9 *Suppose that there is a job L that* RSPT *completes on or before time t, whereas* OPT *completes it after a job J_i in ARRIVE. Then L has $\frac{3}{2}w_i$ of credit that was not taken into account before.*

Proof. In the analyses of previous events, *ARRIVE* was not taken into account when considering the credit of job L. Compared to that analysis, we have that $s^{\text{OPT}}(L)$ increases by at least w_i. Rule C2 implies that L now has $\frac{3}{2}w_i$ more credit than calculated at the previous events. □

Since for a completed job L we have $N_{COM}(L,t) = N_{INT}(L,t) = 0$, we can give this credit of $\frac{3}{2}w_i$ to other jobs, while L still satisfies (6.8).

Suppose that OPT runs at least one large job J' before *ARRIVE*. In this case we will not use any lower bound on the optimal starting time of J' (except that it is at least 0). This enables us to make the following assumption.

Event assumption 3 *If* OPT *runs at least one large job before ARRIVE (i.e., at least one job from the set $\{J^1 = J, J^2, J^3, \ldots\}$), then J is one of these jobs.*

We explain why we can make this assumption without loss of generality. Suppose OPT runs $J' \neq J$ before *ARRIVE*, but not J. By Global assumption 1, J' then has some size $y > x$ where x is the size of J. This can only increase the optimal starting time of jobs in *ARRIVE* compared to the situation where OPT does run J before *ARRIVE*: if OPT runs only J before *ARRIVE*, it can start to run the jobs in *ARRIVE* at time x, whereas with J' it can only start to run them at time $y > x$. A similar statement holds if OPT runs two (or more) large jobs before *ARRIVE*.

Consider an event at time t, where J is interrupted or completed after starting at time s. We will calculate how some relevant variables change from time s to time t in this situation (i.e. OPT runs J after *ARRIVE* and J' before it). First of all, we have $s_t^{\text{OPT}}(J) \geq s_s^{\text{OPT}}(J) + W$, so by Rule C2,

$$K_t(J) \geq K_s(J) + \frac{3}{2}W \quad \text{(minus interrupt-delay } t - s \text{ in case of an interruption)} \tag{6.9}$$

Second, $s_t(J') = s_s(J') + W$ (plus interrupt-delay) so by Rule C1,

$$K_t(J') = K_s(J') - W \quad \text{(minus interrupt-delay)}. \tag{6.10}$$

Third, it can be seen that

$$N_{INT}(J',t) \leq N_{INT}(J',s) + \frac{1}{2}W. \tag{6.11}$$

We briefly explain why. If there is an interruption at time t, then since $y > x$ there is at least one job (J) between *ARRIVE* and J' in RSPT's schedule, and (6.11) follows from (6.6). If there is a completion at time t, then $t \geq x > w_k$, so by (6.6) we have $N_{INT}(J',t) \leq w_k + \frac{1}{2}(w_{k-1} + w_{k-2}) - t \leq \frac{1}{2}(w_{k-1} + w_{k-2}) \leq \frac{1}{2}(W - w_k) \leq N_{INT}(J',s) + \frac{1}{2}(W - w_k)$.

Now, if OPT runs J before *ARRIVE* instead of J', we have $K_t(J) = K_s(J)$ instead of (6.9). However, (6.10) and (6.11) still hold. Therefore, it is sufficient to analyze the case where OPT runs J before *ARRIVE* (i.e. at time 0), and ensure that the invariant is maintained. Then, we can switch J and J', and transfer an additional $\frac{3}{2}W$ worth of credit (that we have by (6.9), but did not use) from J to J'. We will not make this explicit anymore in the rest of the paper but simply make Event assumption 3.

Lemma 6.10 *The condition (6.2) can be maintained throughout the execution of σ.*

Proof. Using Lemma 6.6 we can calculate valid initial values $s_t^{\mathrm{OPT}}(J)$ for any arriving job J. The only modification we will make in later events, is that for any $J(I)$-large job J' that OPT completes after *ARRIVE*(I), we increase $s_t^{\mathrm{OPT}}(J')$ by $W_k(I)$. (Here we will use Event assumption 3.) Since the set *ARRIVE*(I) was not taken into consideration when $s_t^{\mathrm{OPT}}(J')$ was originally determined, the resulting bound is still valid, so (6.2) holds. □

Lemma 6.11 *The large jobs that both* OPT *and* RSPT *complete after ARRIVE besides J satisfy (6.8).*

Proof. Consider such a job J^i. It receives extra credit of $\frac{1}{2}W$ by Rules C1 and C2, since both $s(J^i)$ and $s^{\mathrm{OPT}}(J^i)$ increase by W. It possibly loses some credit if there was an interruption. However, Table 6.1 combined with (6.5) ensures that any job that does not have enough credit to pay for the interruption of J gets the remainder from J_1.

Note that $N_{COM}(J^i,t) \le N_{COM}(J^i,s)$ for all $s \ge t$. Moreover, exactly as above (after Event assumption 3), we have $N_{INT}(J^i,t) \le N_{INT}(J^i,s) + \frac{1}{2}W$. Thus the extra credit of $\frac{1}{2}W$ that J^i receives ensures that it satisfies (6.8). □

By the results in this section, in the remainder of the paper it is sufficient to check (or ensure) that J, the jobs in *ARRIVE*, and any other jobs that OPT has completed but RSPT has not satisfy (6.8). Moreover, by Event assumption 3 the optimal schedule is defined by nothing but the *number* of large jobs that it runs before *ARRIVE* (and the size of those jobs, if there is more than 1).

6.5 Interruptions

Consider an interruption at time r_1 of a job J that started at time s. The interrupt-delay associated with this interruption is $r_1 - s$ for all jobs that were already in Q at time s. The small job-delay of this event of the jobs that OPT completes before *ARRIVE*, and RSPT does not, is $W = \sum_{i=1}^k w_i$.

The last event before this one was the start of J, at time s. No jobs were completed since then by RSPT. We divide the interruptions into three types, based on the number of large jobs OPT runs before *ARRIVE*.

Large jobs before *ARRIVE* by OPT	0	1	2
Lemma	6.12	6.13	6.14

We need to distinguish between the cases $f = 1$ and $f > 1$. First suppose $f = 1$, i. e. OPT runs the jobs from *ARRIVE* in the same order as RSPT. The credit reassignments in this case take place in four steps.

1. On arrival of J_1, the jobs $J, J_2, \ldots, J_k$ are shifted. However, for the moment we keep the order of those jobs the same (as in the situation where J_1 does not arrive). We reassign credit from J_1 to J so that its credit remains constant. (Some jobs can have negative credit in this step.)

2. We reorder the jobs so that the order is now $J_1, \ldots, J_k, J$. However, the credits of the jobs stay “in the same place”, so that e. g. J_2 now has the credit that J had in Step 1.

3. We calculate the extra credit that this reordering generates. If a completion is now δ time earlier, there is δ more credit available by Rule 1.

4. We reassign credits to make sure all waiting jobs satisfy (6.8). (This step is not always required.)

A graphical representation of the first three steps of this procedure can be seen in Figure 6.2.

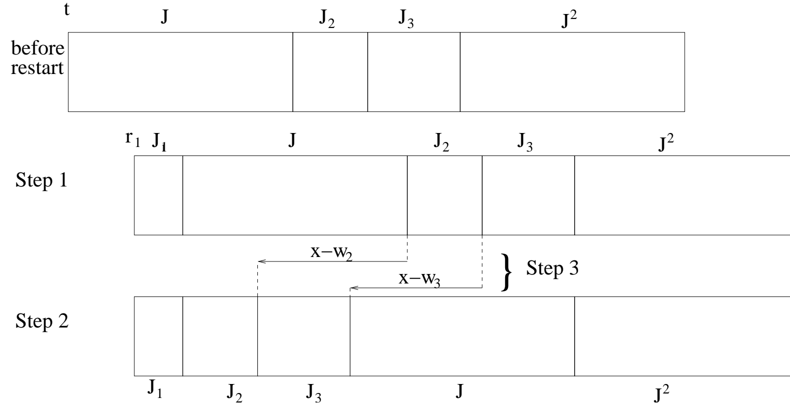


Figure 6.2: Credit transfers

In case $f > 1$, we proceed similarly. However, in the first step we consider different orders for the jobs (which will be described at the time), instead of the order described above. We then put the jobs in the order they will be executed by RSPT in Step 2 and continue as above.

Lemma 6.12 *If* RSPT *interrupts a job J at time r_1, and* OPT *runs no large jobs before ARRIVE, and $s \geq x/2$, and STATIC holds, and the invariant held at time s, then it holds at time r_1.*

Proof. Case 1. $f = 1$. Only in this very first case will we not use the procedure outlined above, and instead calculate the credits for the proper order directly. We begin by showing that J_1 still satisfies (6.8) after giving credit to J and J-large jobs as described in Table 6.1. We have initially $K(J_1) = \frac{1}{2}(r_1 + w_1)$. Also, $N_{INT}(J_1, r_1) = 0$ and $N_{COM}(J_1, r_1) \leq \frac{1}{2}w_1$ by definition. We need to check every column in Table 6.1.

Suppose $s < r_1 \leq x$. Since $s \geq x/2$, we have $r_1 \leq \frac{2}{3}(s+x) \leq 2s$. Then $\frac{1}{2}(r_1 + w_1) - (r_1 - s) = \frac{1}{2}w_1 + s - \frac{1}{2}r_1 \geq \frac{1}{2}w_1$.

Suppose $s \leq x < r_1$. Since $r_1+w_1 \leq \frac{2}{3}(s+x)$, then using Table 1, we have that J_1 is left with credit of $\frac{1}{2}(r_1+w_1)-(2r_1+w_1-s-x) = s+x-\frac{3}{2}r_1-\frac{1}{2}w_1 \geq s+x-\frac{3}{2}(\frac{2}{3}(s+x)-w_1)-\frac{1}{2}w_1 = w_1$.

Suppose $x < s < r_1 \leq \frac{3}{2}x$. J_1 is left with $\frac{1}{2}(r_1+w_1)-2(r_1-s)-w_1 = 2s-\frac{3}{2}r_1-\frac{1}{2}w_1 \geq w_1$, since $r_1 + w_1 \leq \frac{4}{3}s$.

Suppose $x < s \leq \frac{3}{2}x < r_1$. We take $2(r_1 - s) + w_1 + 2(r_1 - \frac{3}{2}x)$ out of J_1. J_1 still has $\frac{1}{2}(r_1 + w_1) - 4r_1 + 2s + 3x - w_1 \geq -\frac{7}{2} \cdot \frac{2}{3}(s + x) + 3w_1 + 2s + 3x = -\frac{1}{3}s + \frac{2}{3}x + 3w_1 > 3w_1$ of credit.

Suppose $\frac{3}{2}x < s$. We take $4(r_1-s)+w_1$ out of J_1, leaving it with $\frac{1}{2}(r_1+w_1)-4r_1+4s-w_1 \geq 4s - \frac{7}{3}(s + x) + 3w_1 = \frac{5}{3}s - \frac{7}{3}x + 3w_1 > 3w_1$.

	1	**2**	**3**	**4**	final
J_1	$\frac{1}{2}w_1 + \frac{3}{2}D_2$	J_1	0	$-\frac{3}{2}D_2$	$\frac{1}{2}w_1$
J_f	$\frac{3}{2}r_f + \frac{1}{2}w_f - r_1$	J_2	0	$\frac{3}{2}D_2$	$\frac{1}{2}w_f$
$i = 2, \ldots, f-1:$					
J_i	$\frac{1}{2}(r_f + w_f + W_i) - D_1$	J_{i+1}	$w_f - w_i$	0	$\frac{r_f+3w_f+W_{i-1}-w_i}{2} - D_1$
$i = f+1, \ldots, k:$					
J_i	$\frac{1}{2}(r_f + W_i) - D_1$	J_i	0	0	$\frac{1}{2}(r_f + W_i) - D_1$
J	$N_{COM}(J, s) + \frac{1}{2}W$	J	0	0	$N_{COM}(J, r_1) + N_{INT}(J, r_1)$

Table 6.3: Credits in Lemma 6.12

In all cases, $K(J_1)$ satisfies (6.8). For $2 \leq i \leq k$, we have $K(J_i) = \frac{1}{2}(r_1 + W_i)$ so that we are done by Lemma 6.7, Case 1. For J, we have that $s^{\mathrm{OPT}}(J)$ increases by $\frac{1}{2}W$ (see proof of Lemma 6.10). Note that $\frac{1}{2}W \geq w_k + \frac{1}{2}w_{k-1} + \frac{1}{2}w_{k-3} - r_1 \geq N_{INT}(J, r_1)$, since $r_1 > s \geq x/2 > w_k/2$. We also have $N_{COM}(J, r_1) \leq N_{COM}(J, s)$, so J still satisfies (6.8).

Case 2. $f > 1$. We define $D_1 = r_1 - r_f > 0$ (if $r_1 = r_f$, then by Lemma 6.6 OPT runs the jobs in order of increasing size after time r_1 and hence $f = 1$) and $D_2 = (r_f + w_f) - (r_1 + w_1) > 0$ (if $D_2 = 0$, either J_f would have already caused a restart before time r_1, or J_1 would not have caused a restart).

As in Case 1, we begin by checking the credit of J_1. In this case, initially, $K(J_1) = \frac{3}{2}(r_f + w_f) + \frac{1}{2}w_1 - r_1$ since $s^{\mathrm{OPT}}(J_1) = r_f + w_f$ and $s(J_1) = r_1$. We take an extra w_1 out of $K(J_1)$ for J_2.

Suppose $r_1 \leq x$. Then $\frac{3}{2}(r_f + w_f) - \frac{1}{2}w_1 - 2r_1 + s = \frac{3}{2}D_2 + w_1 - \frac{1}{2}r_1 + s > w_1$. We have used $r_1 \leq \frac{2}{3}(s + x) < 2s$ which holds since $s \geq \frac{2}{3}x$.

Suppose $s \leq x < r_1$. In this case we take $2r_1 + 2w_1 - s - x$ of credit out of J_1. Since $r_1 + w_1 \leq \frac{2}{3}(s + x)$, we have that J_1 is left with credit of $\frac{3}{2}(r_f + w_f - w_1) - 3r_1 + s + x = \frac{3}{2}D_2 - \frac{3}{2}r_1 + s + x \geq \frac{3}{2}D_2 + \frac{3}{2}w_1$.

Suppose $x < s < r_1 \leq \frac{3}{2}x$. We take $2(r_1 - s) + 2w_1$ of credit out of J_1. Since $r_1 + w_1 \leq \frac{4}{3}s$ in this case, again J_1 is left with at least $\frac{3}{2}w_1 + \frac{3}{2}D_2$.

Suppose $x < s \leq \frac{3}{2}x < r_1$. We take $2(r_1 - s) + 2w_1 + 2(r_1 - \frac{3}{2}x)$ out of J_1. J_1 still has $\frac{3}{2}(r_f + w_f - w_1) - 5r_1 + 2s + 3x \geq 3x + 2s - \frac{7}{2}r_1 + \frac{3}{2}D_2 \geq x - \frac{1}{2}r_1 + \frac{3}{2}D_2 + 3w_1 \geq 3\frac{1}{2}w_1 + \frac{3}{2}D_2$ of credit, using $3(r_1 + w_1) \leq 2(s + x)$.

Suppose $\frac{3}{2}x < s$. We take $4(r_1 - s) + 2w_1$ out of J_1, leaving it with $\frac{3}{2}(r_f + w_f - w_1) - 5r_1 + 4s \geq 4s - 3\frac{1}{2}r_1 + \frac{3}{2}D_2 \geq 4s - \frac{7}{2}(\frac{2}{3}(s + x) - w_1) + \frac{3}{2}D_2 \geq \frac{5}{3}s - \frac{7}{3}x + \frac{3}{2}D_2 + \frac{7}{2}w_1 > \frac{7}{2}w_1 + \frac{3}{2}D_2$ since $\frac{5}{3}s \geq \frac{5}{2}x > \frac{7}{3}x$.

In all cases, (6.8) holds for J_1. For the other jobs, we use Table 6.3. In this table, the column marked 1 contains the credits of the jobs assuming the order $J_1, J_f, J_2, \ldots, J_{f-1}, J_{f+1}, \ldots, J_k$, J, and after J_1 has given away credit to J-large jobs as described in Table 1. Column 2 shows the new order of the jobs. The credits stay in the same place, hence e. g. J_2 now has a credit of $\frac{3}{2}r_f + \frac{1}{2}w_f - r_1$. Column 3 shows how much credit is gained by the reordering of the jobs.

Column 4 shows credit transfers; in this case, $\frac{3}{2}D_2$ is transferred from J_1 to J_2. The last column contains the final credit of each job. The numbers above the columns refer to the steps in the procedure described at the start of this section.

We now show that the credit in the last column is sufficient so that all jobs satisfy (6.8).

We begin with job J_2. We have $w_1 = (r_1 + w_1) - r_1 \leq \frac{2}{3}(s + x) - s = \frac{2}{3}x - \frac{1}{3}s \leq x/2$ and $r_1 > x/2$, so J_1 cannot start before time w_1. Hence $N_{INT}(J_2, r_1) = 0$. Also $N_{COM}(J_2, r_1) \leq \frac{1}{2}(w_2 - w_1)$. As in Case 1, we use that $s^{\mathrm{OPT}}(J)$ has increased by $\frac{1}{2}W$. There are two cases.

Suppose $r_1 \geq w_k$. Then $N_{INT}(J, r_1) \leq \frac{1}{2}(w_{k-1} + w_{k-3})$. Therefore J can give $\frac{1}{2}w_k \geq \frac{1}{2}w_f$ of credit to J_2, and still satisfy (6.8). Furthermore, $\frac{3}{2}r_f + \frac{1}{2}w_f - r_1 \geq 0$ since $r_1 \leq 2s \leq 2r_f$, so we are done.

Suppose $r_1 < w_k < x$. If $f = k$, then $r_1 < w_f$ and $\frac{3}{2}r_f - r_1 + \frac{3}{2}D_2 \geq 3r_f + \frac{3}{2}w_f - \frac{5}{2}r_1 - \frac{3}{2}w_1 \geq 0$ since $r_1 + w_1 < \max(2r_f, r_f + w_f)$. If $f < k$, then J_k gives $\frac{1}{2}r_f$ to J_f, and $2r_f - r_1 > 0$. Below we will see that J_k can spare this credit.

For jobs $J_3, \dots, J_f$, we distinguish between two cases. If $r_1 \geq w_{i-1}$, we use $D_1 = w_f - w_1 - D_2 < w_f$ and are done by Lemma 6.7, Case 2. If $r_1 < w_{i-1}$, then we use $w_f - D_1 = w_1 + D_2$. We need to check $\frac{r_f + w_f + W_{i-1} - w_i}{2} + w_1 + D_2 \geq \frac{1}{2}(w_i - w_{i-1}) + \frac{1}{2}W_{i-2} + (w_{i-1} - r_1) = \frac{1}{2}W_i - r_1$. This is equivalent to $\frac{3}{2}(r_f + w_f) \geq w_i$, which holds.

For jobs $J_{f+1}, \dots, J_k$, we have the same two cases. If $r_1 \geq w_{i-1}$, then $K(J_i) \geq \frac{1}{2}(r_f + W_i) - D_1 \geq \frac{1}{2}(w_i - w_{i-1} + W_{i-2})$ (using $D_1 < w_f$), And if $r_1 < w_{i-1}$, then $\frac{1}{2}W_i - D_1 \geq \frac{1}{2}(w_i - w_{i-1} + W_{i-2}) + (w_{i-1} - r_1)$. We use from Lemma 6.7 either Case 2 or Case 3. □

Lemma 6.13 *If* RSPT *interrupts a job J at time r_1, and* OPT *runs one large job before ARRIVE, and STATIC holds, and the invariant held at time s, then it holds at time r_1.*

Proof. We distinguish between three cases depending on s and f. We ignore that OPT has to run the jobs in $BEFORE_s(J)$ too at some point; this can only decrease the optimal cost on the other jobs.

Case 1. $s \leq x/2$. We have $r_1 + w_1 \leq \frac{2}{3}(s + x) \leq x$, so $f = 1$ by Lemma 6.6. Also $b_s(J) \leq s < x$.

We have $s^{\mathrm{OPT}}(J_1) \geq x$ and $D(J_1) \leq r_1 - x$, so $K(J_1) \geq \frac{3}{2}x + \frac{1}{2}w_1 - r_1$. On arrival of J_1, job J is in Step 1 shifted by RSPT by $r_1 + w_1 - s$ time. We take $r_1 + 2w_1 - s$ of credit out of J_1 for J, so that J_1 is left with $\frac{3}{2}x + \frac{1}{2}w_1 - 2(r_1 + w_1) + s \geq \frac{1}{2}w_1 + \frac{1}{6}x - \frac{1}{3}s \geq \frac{1}{2}w_1$, so J_1 satisfies (6.8).

For the other jobs, we refer to Table 6.4. For the Step 1-column, we use (6.3). J_1 satisfies (6.8) by Lemma 6.7, Case 1; J_2 as well, since $N_{COM}(J_2, r_1) \leq N_{COM}(J, s)$ using that $x > w_2$ and $b_s(J) \leq b_{r_1}(J_2)$; the other jobs too by Lemma 6.7, Case 3.

Case 2. $s > x/2$ and $f = 1$. Since $f = 1$, we have in Step 1 of our calculations that $D(J_i) \leq \min(x, r_1)$ for $2 \leq i \leq k$: after time $r_1 + w_1$, RSPT first runs J (of size x) whereas OPT runs $J_2, \dots, J_k$ immediately after J and J_1, at most $\min(x, r_1)$ time earlier. We also have $w_1 \leq \frac{2}{3}(s + x) - r_1 \leq \frac{2}{3}x - \frac{1}{3}s \leq \frac{1}{2}x < s$, so $N_{INT}(J_2, r_1) = 0$.

	1	**2**	**3**	**final**
J_1	$\frac{1}{2}w_1$	J_1	0	$\frac{1}{2}w_1$
J	$N_{COM}(J,s)+w_1$	J_2	0	$N_{COM}(J,s)+w_1$
J_2	$\frac{1}{2}(x+W_2)-r_1$	J_3	$x-w_2$	$\frac{1}{2}(x+W_1-w_2)+(x-r_1)$
	$\vdots$	$\vdots$		
J_k	$\frac{1}{2}(x+W_k)-r_1$	J	$x-w_k$	$\frac{1}{2}(x+W_{k-1}-w_k)+(x-r_1)$

Table 6.4: Credits in Lemma 6.13, Case 1 ($s \le x/2$)

	1	**2**	**3**	**final**
J_1	$\frac{1}{2}w_1$	J_1	0	$\frac{1}{2}w_1$
J	$N_{COM}(J,s)$	J_2	0	$N_{COM}(J,s)$
J_2	$\frac{1}{2}(x+W_2)-x$	J_3	$x-w_2$	$\frac{1}{2}(x+W_1-w_2)$
	$\vdots$	$\vdots$		
J_k	$\frac{1}{2}(x+W_k)-x$	J	$x-w_k$	$\frac{1}{2}(x+W_{k-1}-w_k)$

Table 6.5: Credits in Lemma 6.13, Case 2 ($s > x/2$ and $f = 1$), $r_1 > x$

If $r_1 \le x$, we are done exactly as in Case 1. Otherwise, we can check $K(J_1)$ as in Lemma 6.12, Case 1. For the other jobs, we refer to Table 6.5. Since $r_1 > x > w_i$ for $1 \le i \le k$, it follows immediately from Lemma 6.7, Case 2, that all jobs satisfy (6.8).

Case 3. $s > x/2$ and $f > 1$. Write $\tilde{r} = \max(x, r_f)$. Suppose $r_1 \le x$. Then at time $\tilde{r} \ge r_1$, OPT will run the remaining jobs in order of increasing size by Lemma 6.6. But then $f = 1$, a contradiction. Therefore $r_1 > x$ and $D(J_1) \le r_1 - (\tilde{r} + w_f) < 0$ using Property R3.

Claim: At most two jobs in *ARRIVE* are interruptible.

Proof: Suppose there are three. Then by Property R5 the first (smallest) has size at least $p = (r_1 + w_1)/2 > x/2$, the second one has size at least $p' = (r_1 + w_1 + p)/2 > \frac{3}{4}x$, so the third one has size at least $(r_1 + w_1 + p + p')/2 > x$,a contradiction to Global assumption 2. □

Claim: If some job J_i is interruptible, then 1) either J_{i-1} or J_{i+1} or no job in *ARRIVE*$\backslash\{J_i\}$ is interruptible, 2) $N_{INT}(J_i, r_1) = 0$ and 3) $N_{COM}(J_i, r_1) \le \frac{1}{2}(w_i - W_{i-1})$.

Proof: 1) As the first claim. 2) We have $w_{i-1} - s_{i-1} < 0$ for $i = 1, \ldots, k$. Jobs J_{i-2} and earlier are not interruptible. 3) We have $W_{i-1} < w_{i-1}$, since all jobs in *ARRIVE* start after time $r_1 > x > w_1$ and J_i is interruptible. □

Claim: If J_i is not interruptible, and J_{i-2} is, then $N_{INT}(J_i, r_1) = \max(\frac{3}{2}w_{i-2} - s_{i-2}, 0)$ and $N_{COM}(J, r_1) = 0$.

Proof: We have $w_{i-2} + w_{i-1} > x > w_i$, $r_1 > x > w_1$ and J_{i-4} is not interruptible. □

Since $s^{\text{OPT}}(J_1) \ge \tilde{r} + w_f$, we have $K(J_1) \ge \frac{3}{2}(\tilde{r} + w_f) + \frac{1}{2}w_1 - r_1$. Define $D_2 = (\tilde{r} + w_f) - (r_1 + w_1) > 0$, and $D_1 = r_1 - \tilde{r} > 0$. We will make much use of the following property:

$$w_f - D_2 = (r_1 + w_1) - \tilde{r} \le \frac{2}{3}(s + x) - \tilde{r} \le \frac{2}{3}x - \frac{1}{3}\tilde{r} \le \frac{1}{3}x \le \frac{1}{3}\tilde{r}. \tag{6.12}$$

	J_1	J	J_f
1	$\frac{3}{2}w_1 + \frac{3}{2}D_2$	$N_{COM}(J,s)$	$\frac{\tilde{r}+w_f}{2} - D_1 - x - w_1$
2a	J_1	J	J_2
3a	0	0	0
2b	J_1	J_2	J_3
3b	0	0	$x - w_2$
4	$-\frac{1}{2}D_2$	0	$\frac{1}{2}D_2$
final	$\frac{3}{2}w_1 + D_2$	$N_{COM}(J,s)$	$\frac{1}{2}(w_3 - w_2)$

	$J_i\ (2 \le i \le f-1)$	$J_i(f+1 \le i \le k-1)$	J_k
1	$\frac{\tilde{r}+w_f+W_i}{2} - D_1 - x$	$\frac{\tilde{r}+W_i}{2} - D_1 - x$	$\frac{1}{2}(\tilde{r} + W_k) - D_1 - x$
2a	J_{i+1}	J_i	J_k
3a	$w_f - w_i$	0	0
2b	J_{i+2}	J_{i+1}	J
3b	$x - w_{i+1}$	$x - w_i$	$x - w_k$
4	0	0	0
final	$\frac{\tilde{r}+w_f+W_{i-3}-w_{i-2}}{2} - D_1$	$\frac{\tilde{r}+W_{i-3}-w_i}{2} + D_2 + w_1$	$\frac{\tilde{r}+W_{k-2}-w_k}{2} + D_2 + w_1$

Table 6.6: Credit transfers in Lemma 6.13, Case 3

This property implies $D_1 = w_f - w_1 - D_2 \le w_f - D_2 \le \frac{1}{3}x$ and $D_1 \le w_f$.

We consider the various possibilities for s and r_1 and take credit out of J_1 as described in Table 6.1. The calculations are identical to the ones in Lemma 6.12, Case 2, except that r_f is replaced by $\tilde{r}$ and D_2 is defined as above. It follows that J_1 ends up with credit of at least $\frac{3}{2}(D_2 + w_1)$ and satisfies (6.8).

Credit transfers Since $r_1 > x$, we can again use Case 2 of Lemma 6.7 to check if jobs satisfy (6.8). We transfer credits as in Table 6.6. We now have columns for jobs in stead of rows as before, due to space constraints. Note that in this case, there are two reorderings of the jobs (a and b). Also, there is an additional row 4, indicating credit transfers between jobs. Note that the entries in this row add up to 0. The entries in the last row will be explained below.

J_1 is not interruptible because $w_1 \le \frac{2}{3}(s+x) - s = \frac{2}{3}x - \frac{1}{3}s < \frac{1}{2}x < \frac{1}{2}r_1$. Moreover, $r_1 > x > w_2$, so $N_{INT}(J_2, r_1) = N_{INT}(J_3, r_1) = 0$. For J_2, we have $N_{COM}(J,s) = \frac{1}{2}(x - b_s(J))$, $x > w_2$ and $b_s(J) \le b_{r_1}(J_2)$. Therefore J_2 satisfies (6.8).

For J_3, we have $K(J_3) = \frac{1}{2}(\tilde{r} + w_f) - D_1 - x - w_1 + x - w_2 + \frac{1}{2}D_2 = \frac{1}{2}(\tilde{r} + w_f) + \frac{3}{2}D_2 - w_2 - w_f \ge \frac{1}{2}(\tilde{r} - 2w_2 - w_f) + \frac{3}{2}D_2 = \frac{1}{2}(w_3 - w_2) + \frac{1}{2}(\tilde{r} - w_2 - w_3 - w_f + 3D_2) \ge \frac{1}{2}(w_3 - w_2)$ using (6.12).

For $i = 4, \ldots, f$ we find

$$\begin{aligned} K(J_i) &= \frac{1}{2}(\tilde{r} + w_f + W_{i-2}) - D_1 - x + x - w_{i-1} + w_f - w_{i-2} \\ &\ge \frac{1}{2}(\tilde{r} + w_f + W_{i-3} - w_{i-2}) - D_1 \end{aligned} \qquad (6.13)$$

$$
\begin{aligned}
&= \frac{1}{2}(\tilde{r} + W_{i-3} - w_f - w_{i-2}) + D_2 + w_1 \qquad (6.14)\\
&= \frac{(w_i + W_{i-3} - w_{i-2}) + (\tilde{r} - w_f - w_i + 2D_2)}{2} \geq \frac{w_i + W_{i-3} - w_{i-1}}{2}.
\end{aligned}
$$

If J_i is interruptible, J_{i-2} and earlier jobs are not and we are done. Suppose J_i is not interruptible and $s_{i-2} < \frac{3}{2}w_{i-2}$. (If $s_{i-2} \geq \frac{3}{2}w_{i-2}$, we are done.) Then we use that $s_{i-2} > r_1$ and we have from (6.13)

$$K(J_i) + s_{i-2} > \frac{1}{2}(\tilde{r} + w_f + W_{i-3} - w_{i-2}) + \tilde{r} \geq \frac{1}{2}(\tilde{r} - w_{i-2} + W_{i-3}) + \frac{3}{2}w_{i-2}.$$

For J_{f+1}, the calculations are similar, but from (6.14) we now derive $K(J_{f+1}) \geq \frac{1}{2}W_{f-1} + \frac{1}{2}(w_{f+1} - W_f)$ and $K(J_i) \geq \frac{1}{2}W_{f-1}$ (using (6.12)). Thus there is sufficient completion-credit, and the case $s_{f-1} < \frac{3}{2}w_{f-1}$ is handled as above.

For $i = f+2, \ldots, k$ we have $K(J_i) \geq \frac{1}{2}(\tilde{r} + W_{i-1}) - D_1 - x + x - w_{i-1} \geq \frac{1}{2}(\tilde{r} + W_{i-2} - w_{i-1}) - D_1$. There are five cases.

1) If $w_{i-2} \geq \frac{3}{2}s_{i-2}$, then J_{i-2} is interruptible and by the third claim above it is sufficient to show $K(J_i) \geq \frac{3}{2}w_{i-2} - s_{i-2}$. Since we have $K(J_i) + r_1 \geq \frac{1}{2}(\tilde{r} + W_{i-2} - w_{i-1}) + \tilde{r} \geq \frac{1}{2}(\tilde{r} + W_{i-3} - w_{i-1}) + \frac{3}{2}w_{i-2}$, this implies $K(J_i) \geq \frac{3}{2}w_{i-2} - r_1 \geq \frac{3}{2}w_{i-2} - s_{i-2}$.

2) Otherwise, $N_{INT}(J_i, r_1) \leq 2w_{i-4} - r_1$. In fact, if $r_1 < 2w_{i-4}$ then $K(J_i) + r_1 \geq \frac{1}{2}(\tilde{r} + W_{i-2} - w_{i-1}) + \tilde{r} \geq \frac{1}{2}W_{i-2} + \tilde{r} \geq \frac{3}{2}w_{i-4} + \tilde{r} > 2w_{i-4}$, which is sufficient since $N_{COM}(J_i, r_1) = 0$.

In the remaining cases, $N_{INT}(J_i, r_1) = 0$ and $N_{COM}(J_i, r_1) \leq \frac{1}{2}(w_i - W_{i-1})$.

3) If $i \geq f+3$, then $\frac{1}{2}W_{i-2} - D_1 \geq \frac{1}{2}w_{i-3} + \frac{1}{2}w_{i-2} - D_1 \geq w_f - D_1 = w_1 + D_2 \geq 0$, so $K(J_i) \geq \frac{1}{2}(\tilde{r} - w_{i-1})$ and we are done.

We are left with J_{f+2}. We have $K(J_{f+2}) \geq \frac{1}{2}(\tilde{r} + W_{f-1} - w_f - w_{f+1}) + D_2 + w_1$. Clearly $K(J_{f+2}) \geq \frac{1}{2}(w_{f+2} - W_{f+1})$, so it is sufficient to show $K(J_{f+2}) \geq 0$.

4) If $w_f \geq \frac{2}{3}x$, then $D_2 \geq \frac{1}{3}\tilde{r}$ and thus $K(J_{f+2}) \geq \frac{1}{2}(w_{f+2} + W_{f-1} - w_{f+1}) + \frac{1}{2}(\tilde{r} - (w_f - D_2)) - \frac{1}{2}(w_{f+2} - D_2) \geq \frac{1}{2}(w_i + W_{i-3} - w_{i-1})$.

5) If $w_f < \frac{2}{3}x$, then also $w_2 < \frac{2}{3}x$. For this particular case only, we consider when OPT runs the jobs in $BEFORE_s(J)$. If any job in $BEFORE_s(J)$ is completed after J_f, then by Lemma 6.9 there is extra credit available of $\frac{3}{2}w_f$. We give this to J_{f+2} which then has credit of at least $\frac{1}{2}(w_i + W_{i-3} - w_{i-1}) + \frac{1}{2}(\tilde{r} - w_i) \geq \frac{1}{2}(w_i + W_{i-3} - w_{i-1})$.

If all jobs in $BEFORE_s(J)$ are completed before J_f, then the credit of J_f is $\frac{3}{2}b_s(J)$ larger than previously calculated. We give this to J_2. Then $K(J_2) \geq \frac{1}{2}x$. However, $w_2 < \frac{2}{3}x$, so we take $\frac{1}{6}x$ out of the credit of J_2 again and give it to J_{f+2}. Then $K(J_{f+2}) \geq \frac{1}{2}(\tilde{r} + W_{f-1} - w_{f+1}) + \frac{1}{6}x + \frac{1}{2}w_f - D_1 \geq \frac{1}{2}(\tilde{r} + W_{f-1} - w_{f+1})$ since $D_1 \leq \frac{1}{3}x$ and $D_1 \leq w_f$.

For J, we calculate as for J_{f+1} in case $k = f$ and as for J_{f+2} and higher in case $k > f+1$. (The calculations for $J_{f+1}, \ldots, J_k$ hold for any job size at most x, so they also hold when applied to J.) □

Lemma 6.14 *If* RSPT *interrupts a job J at time r_1, and* OPT *runs at least two large jobs before ARRIVE, and STATIC holds, and the invariant held at time s, then it holds at time r_1.*

Proof. Since RSPT only interrupts J before time $2x$, we have $f = 1$: OPT starts to run the jobs in *ARRIVE* after RSPT does, and will use the optimal order for them by Lemma 6.6. That Lemma also implies that OPT runs not more than two large jobs before *ARRIVE*. If $s \le x/2$, we have $r_i \le x$ for all jobs $J_i \in$ *ARRIVE*. Then, again by Lemma 6.6, we may assume OPT runs only one large job before *ARRIVE*: a contradiction. Hence $x/2 < s < 2x$ and $r_1 > x$.

Suppose $\ell_s(J) < x$. In this case, we ignore that OPT has to run the job of this size too at some time. This can only help OPT.

Credit of J_1 and large jobs We define $D_1 = -D(J_1) = x + x_2 - r_1 \ge \frac{4}{3}x - \frac{2}{3}s + w_1$. We have $K(J_1) = \frac{3}{2}(x + x_2) + \frac{1}{2}w_1 - r_1$. We consider the various possibilities for s and r_1 and take credit out of J_1 as described in Table 6.1, and an extra w_1 for the small-job delay of J^2. By these reassignments, and because J and J^2 satisfied (6.8), we have in Step 1 that $K(J) \ge \frac{1}{2}(x - b_s(J))$ and $K(J^2) \ge \frac{1}{2}(x_2 - x) - W_k + w_1$.

Suppose $s \le x$. Then $r_1 \le \frac{2}{3}(s + x) \le \frac{4}{3}x$. We take $2(r_1 + w_1) - s - x$ of credit out of J_1 to give to J and J^2, and we let it keep $\frac{1}{2}w_1$ for itself. We denote the remainder, which will be given to J^2, by K_1. We have $r_1 \le \frac{2}{3}(x + s) - w_1$, so in this case $K_1 = \frac{3}{2}(x + x_2) - 3r_1 - 2w_1 + s + x \ge \frac{1}{2}x + \frac{3}{2}x_2 - s + w_1 \ge x_2 + w_1$.

Now suppose $x < s < r_1 \le \frac{3}{2}x$. In this case we need $2(r_1 + w_1) - 2s$ for J and J^2. Hence $K_1 = \frac{3}{2}(x + x_2) - 3r_1 - 2w_1 + 2s \ge \frac{3}{2}(x + x_2) - 2x + w_1 \ge x_2 + w_1$.

Thirdly, suppose $x < s \le \frac{3}{2}x < r_1$. Now the required credit is in total $2(r_1 - s + w_1) + 2(r_1 - \frac{3}{2}x)$. Hence $K_1 \ge \frac{3}{2}(x + x_2) - \frac{4}{3}s - \frac{1}{3}x + 3w_1 \ge \frac{2}{3}x + 3w_1$.

Finally, if $\frac{3}{2}x < s < r_1$ the jobs require $4(r_1 - s) + w_1$, and again $K_1 \ge \frac{2}{3}x + 3w_1$.

Credit transfers We transfer credits as in Table 6.7.

For $2 \le i \le k$, we have $D(J_i) \le (r_1 + x + W_{i-1}) - (x + x_2 + W_{i-1}) = r_1 - x_2$ in Step 1. Using (6.3) we have $K(J_i) \ge \frac{1}{2}(x + x_2 + W_i) - (r_1 - x_2) \ge \frac{1}{2}W_i + D_1$ for $2 \le i \le k$.

J_1 is not interruptible by Property R5 since $r_1 \ge \max(x, s)$ and $r_1 + w_1 \le \frac{2}{3}(s + x) \le \frac{4}{3}\max(s, x)$, hence $N_{INT}(J_i, r_1) = 0$ for $1 \le i \le 3$. Furthermore, $D_1 = x + x_2 - r_1$ is an upper bound for the total amount of future interrupt-delays caused by interruptions of jobs before J^2 that have arrived so far.

Suppose $k \le 2$. In this case we do not use the table. If $k = 1$, we are done immediately. If $k = 2$, then $N_{INT}(J, r_1) = 0$ and $K(J) = \frac{1}{2}W_2 + D_1 + x - w_2 = \frac{1}{2}(x - w_2 + w_1) + \frac{1}{2}x + D_1$. Thus J can give $\frac{1}{2}x + D_1$ to J^2, and J_1 can give an additional $K_1 \ge \frac{1}{2}x$ to J^2. Then J^2 receives in total $x + D_1 \ge w_2 + D_1$, and it received w_1 already from J_1 at the start. Thus J^2 satisfies (6.8). For job J_2, we have $K(J_2) = N_{COM}(J, s) = \frac{1}{2}(x - b_s(J))$, $x \ge w_2$ and $b_{r_1}(J_2) \ge b_s(J)$, so $K(J_2) \ge N_{COM}(J_2, r_1)$.

Suppose $k \ge 3$. In this case we use Table 6.7. J_3 can give D_1 away because $N_{INT}(J_3, r_1) = 0$. We distinguish between two cases: $r_1 \le \frac{3}{2}x$ and $r_1 > \frac{3}{2}x$.

Case 1. $r_1 \le \frac{3}{2}x$: The jobs $J_4, \ldots, J_k, J, J^2$ have sufficient interrupt-credit because they have D_1. Since $x > w_i$ for $i = 4, \ldots, k$, these jobs also have sufficient completion-credit. J_2 satisfies (6.8) as above. Hence we only need to show that the total transferred credit in Column 4 is at

	1	**2**	**3**	**4**	final
J_1	$\frac{1}{2}w_1 + K_1$	J_1	0	$-K_1$	$\frac{1}{2}w_1$
J	$N_{COM}(J,s)$	J_2	0	0	$N_{COM}(J,s)$
J_2	$\frac{1}{2}W_2 + D_1$	J_3	$x - w_2$	$-\frac{x+w_1}{2} - D_1$	$\frac{1}{2}(x - w_2)$
$i = 3, \ldots, k-1:$					
J_i	$\frac{1}{2}W_i + D_1$	J_{i+1}	$x - w_i$	$-\frac{1}{2}(x + w_{i-1})$	$\frac{x+W_{i-2}-w_i}{2} + D_1$
J_k	$\frac{1}{2}W_k + D_1$	J	$x - w_k$	$-\frac{1}{2}(x + w_{k-1})$	$\frac{x+W_{k-2}-w_k}{2} + D_1$
J^2	$N_{COM}(J^2,s) - W_k + w_1$	J^2	0	$+W_k + D_1$	$N_{COM}(J^2,s) + D_1$

Table 6.7: Credit transfers in Lemma 6.14 ($r_1 \leq \frac{3}{2}x$, $k \geq 3$)

most 0, i. e. all the credit given to J^2 is actually available. To see this, note that $\frac{1}{2}(x + w_{i-1}) \geq w_{i-1}$ for $2 \leq i \leq k$. Furthermore, if $r_1 \leq \frac{3}{2}x$ (shown in the table), we have $K_1 \geq x_2 \geq w_k$. The additional D_1 from J_3 completes the credit given to J^2.

Case 2. $r_1 > \frac{3}{2}x$: Denote the jobs that RSPT starts to run at time r_1 alternatively by $J'_1, \ldots, J'_{k'}$, then by (6.5) we have $N_{INT}(J'_i, r_1) \leq \frac{1}{2}w'_{i-4}$. We take D_1 from J_3 as before and also D_1 from the very next job J'_4 ($J'_4 \in \{J_4, J\}$). We can do that because $N_{INT}(J'_4, r_1) = 0$. This makes in total again at least $W_k + D_1$ to give to J_2, since in this case $K_1 + D_1 \geq x_2 \geq w_k$. Giving it to J^2 again ensures that J^2 satisfies (6.8).

Now suppose $\ell_s(J) \geq x$. In this case OPT runs only one more large job than RSPT before *ARRIVE*. Hence in this case, J^2 does not have small-job delay and we are done after Step 3 in the table: all jobs already satisfy (6.8). □

6.6 Job completions

We divide the job completions into cases based on how many large jobs OPT runs before *ARRIVE*. The case where no jobs arrived at all while RSPT was running J is treated separately in Lemma 6.15. An overview can be found in the following table.

Large jobs before *ARRIVE* by OPT	0	1	2	More than 2
Lemma	6.16	6.17	6.18	6.19, 6.22

Remember that to calculate the initial credit of jobs, we will use Event assumption 2. Since all jobs in *ARRIVE* are smaller than x, and complete after J, we have $N_{COM}(J_i, t) = 0$ for $J_1, \ldots, J_k$. We use again Event assumption 3.

Lemma 6.15 *If* RSPT *completes a job J and no jobs arrived while it was running J, and STATIC holds, and the invariant held at time s, then it holds at time t.*

Proof. RSPT now starts the run the smallest available job at the time that was calculated in the analysis of the most recent event. Hence, for the remaining jobs the situation (and the credit)

does not change. The completed job has nonnegative credit. □

Lemma 6.16 *If* RSPT *completes a job J without interruptions and* OPT *does not run any large jobs before ARRIVE, and STATIC holds, and the invariant held at time s, then it holds at time t.*

Proof. We use similar tables as in Section 6.5, starting with a job order for which it is easy to calculate the credits and then reordering the jobs. Here we ignore that RSPT starts to run the jobs already at time s and not only at time r_f. This gives us a lower bound for the amount of credit that is actually available.

Applying Lemma 6.9 repeatedly, we have that there is $\frac{3}{2}W$ of credit available. We give this to J. We consider first the credit of the jobs if RSPT would run the jobs in the same order as OPT, and starting at time r_f. Then J starts W time later than calculated at the previous event, and thus only has $\frac{1}{2}W$ left of the $\frac{3}{2}W$ that it just received. In Step 1, we have that RSPT starts each job at the same time as OPT starts it. See Table 6.8. We use (6.3).

	1	**2**	**3**	**final**
J_f	$\frac{1}{2}(r_f + w_f)$	J	$-(x - w_f)$	0
$J_i\ (i = 1, \ldots, f-1)$	$\frac{1}{2}(r_f + w_f + W_i)$	J_i	$-(x - w_f)$	$\frac{1}{2}W_i$
$J_i\ (i = f+1, \ldots, k)$	$\frac{1}{2}(r_f + W_i)$	J_{i-1}	$-(x - w_i)$	$\frac{1}{2}(W_i - w_f)$
J	$N_{COM}(J, s) + \frac{1}{2}W$	J_k	0	$N_{COM}(J, s) + \frac{1}{2}W$

Table 6.8: Credits in Lemma 6.16

We use in this table that $\frac{1}{2}(r_f + w_f) - (x - w_f) = \frac{3}{2}w_f - x + \frac{1}{2}r_f \geq s - r_f \geq 0$, which holds since $\frac{3}{2}w_f \geq s + x - \frac{3}{2}r_f$. For $J_f, \ldots, J_{k-1}$ we also use that $x - w_i \leq x - w_f$. Note that $W_i - w_f \geq W_{i-1}$ for $i > f$. Hence all the jobs in Table 6.8 satisfy (6.8) by Lemma 6.7, Case 1. Note that this proof also holds for $f = 1$. □

Lemma 6.17 *If* RSPT *completes J without interruptions and* OPT *runs one large job before ARRIVE, and STATIC holds, and the invariant held at time s, then it holds at time t.*

Proof. If $s \leq x/2$, the jobs in *ARRIVE* start within a factor of $\frac{3}{2}$ of their optimal starting time (and run in the best possible order), so that the credit of J_i is at least $\frac{1}{2}\sum_{j=1}^{i} w_i$ by (6.3): all jobs in *ARRIVE* satisfy (6.8) by Lemma 6.7. The same thing holds if $s \geq 2x$.

Suppose $x/2 < s < 2x$. This implies $N_{INT}(J_i, t) \leq 2w_{i-4} - t \leq \frac{1}{2}w_{i-4}$ for $4 \leq i \leq k$ and $N_{INT}(J_i, t) = 0$ for $1 \leq i \leq 3$. Recall that $N_{COM}(J_i, t) = 0$ for all $J_i \in$ *ARRIVE*.

Suppose $1 \leq f < k$. We define $v_f = s^{\text{OPT}}(J_1) - \frac{2}{3}(s + x) > 0$ and $\tilde{s} = \max(s, x + b_s(J))$. Suppose first that OPT runs the jobs in $BEFORE_s(J)$ before *ARRIVE*, then $s^{\text{OPT}}(J_1) \geq \tilde{s} + w_f$. We have

$$K(J_f) \geq \frac{1}{2}(\tilde{s} + w_f) - (s + x - \tilde{s}) = \frac{3}{2}\tilde{s} + \frac{1}{2}w_f - (s + x) = \frac{3}{2}v_f - w_f.$$

If $s \leq x + b_s(J)$, then $w_f - v_f \leq \frac{2}{3}(s + x) - \tilde{s} \leq \frac{2}{3}(x + b_s(J) + x) - (x + b_s(J) = \frac{1}{3}(x - b_s(J))$. If $s \geq x + b_s(J)$, then $w_f - v_f \leq \frac{2}{3}x - \frac{1}{3}s \leq \frac{1}{3}(x - b_s(J))$ as well. Using this bound, we transfer

credits as in Table 6.9. It can be seen that the entries in Column 4 add up to at most 0, and that all jobs satisfy (6.8).

	1	**2**	**3**	**4**	**final**
J	$\frac{1}{2}(x - b_s(J))$	J	0	$-\frac{1}{2}(x - b_s(J))$	0
J_f	$\frac{3}{2}v_f - w_f$	J_1	0	$\frac{1}{3}(x - b_s(J))$	$\frac{1}{2}v_f$
J_1	$\frac{3}{2}v_f - w_f + \frac{1}{2}w_1$	J_2	$w_f - w_1$	$\frac{1}{2}w_1$	$\frac{3}{2}v_f$
$i = 2, \ldots, f:$					
J_i	$\frac{3}{2}v_f - w_f + \frac{1}{2}W_i$	J_{i+1}	$w_f - w_i$	$\frac{1}{2}(w_i - w_{i-1})$	$\frac{3}{2}v_f + \frac{1}{2}W_{i-2}$
J_{f+1}	$\frac{3}{2}v_f - w_f + \frac{W_{f+1}-w_f}{2}$	J_{f+1}	0	$\frac{x-b_s(J)}{6} - \frac{1}{2}w_{f-1}$	$v_f + \frac{1}{2}W_{f-2}$
$i = f+2, \ldots, k:$					
J_i	$\frac{3}{2}v_f - w_f + \frac{W_i-w_f}{2}$	J_i	0	0	$\frac{3}{2}v_f + \frac{1}{2}W_{i-3}$

Table 6.9: Credit transfers in Lemma 6.17 $(1 \leq f < k)$

If $f = k$, then we cannot take $\frac{1}{2}w_{f-1}$ of credit out of J_{f+1} because there is no such job. However, we can now give $\frac{x-b_s(J)}{6}$ from J to J_f instead of to J_{f+1}, and $\frac{x-b_s(J)}{6} \geq \frac{1}{2}(w_f - v_f) \geq \frac{1}{2}(w_{f-1} - v_f)$.

Finally, consider the case where OPT does not run all jobs in $BEFORE_s(J)$ before all the jobs in *ARRIVE*. Then OPT runs one job in $BEFORE_s(J)$ in particular after job J_f, which implies that there is an additional $\frac{3}{2}w_f$ of credit available by Lemma 6.9. We can give w_f to J_1 and $\frac{1}{2}w_f$ to J_{f+1} (instead of giving those jobs credit from J). For the other jobs, we can still transfer credits as in Table 6.9. If $f = k$, we give $\frac{3}{2}w_f = \frac{3}{2}w_k$ to J_1. □

Corollary 6.3 *If* RSPT *completes J without interruptions and* OPT *runs one large job before ARRIVE, and STATIC holds, the jobs in ARRIVE need to receive at most an additional $\frac{3}{2}(w_f - v_f) \leq \frac{1}{2}(x - b_s(J))$ of credit in total in order to satisfy (6.8), where $v_f = s^{\mathrm{OPT}}(J_1) - \frac{2}{3}(s + x)$.*

Lemma 6.18 *If* RSPT *completes J without interruptions and* OPT *runs two large jobs before ARRIVE, and STATIC holds, and the invariant held at time s, then it holds at time t.*

Proof. If $\ell_s(J) \geq x$, there is nothing to prove since the same number of jobs is delayed by RSPT and by OPT, and RSPT starts the jobs in *ARRIVE* at most a factor of $3/2$ after OPT starts them since $s \geq x$. Hence, the jobs in *ARRIVE* satisfy (6.8) and the remaining large jobs gain credit by Rule C2 and still satisfy (6.8). (We can assume OPT runs the same two large jobs before *ARRIVE* as RSPT, similarly to Event assumption 3.)

Suppose $\ell_s(J) < x$. Denote the largest of the two jobs that OPT completes before *ARRIVE* by J^2. We distinguish between the cases $s \leq x_2$ and $s > x_2$.

Case 1. $s \leq x_2$. Then $K(J_i) \geq \frac{1}{2}(x+x_2+W_i)$. Note that $N_{COM}(J_i, t) = 0$, and $N_{INT}(J_i, t) \leq \frac{1}{2}W_{i-2}$. We let each job J_i keep $\frac{1}{2}W_{i-1}$ and give $\frac{1}{2}(x + x_2 + w_i) > \frac{3}{2}w_i$ to J^2. This is sufficient to both pay for the small job-delay of J^2, which is W, and to add $\frac{1}{2}W$ for $K_{INT}(J^2)$, which ensures $K(J^2) \geq N_{COM}(J^2, t) + N_{INT}(J^2, t)$.

Case 2. $s > x_2$. Then the jobs in *ARRIVE* are not interruptible, hence $N_{INT}(J_i, t) = 0$ and $N_{COM}(J_i, t) = 0$ for all jobs $J_i \in$ *ARRIVE*. Therefore, $N_{INT}(J_2, t) = 0$. Consider the set $BEFORE_s(J)$ of jobs that RSPT already completed, and suppose OPT completes all these jobs before J_k.

Then we have $K(J_i) \geq \frac{3}{2}(\max(s, x + x_2)) + \frac{1}{2}W_i - (s + x) \geq \frac{1}{2}W_i$ for $i = 1, \ldots, k-1$ and $K(J_k) \geq \frac{3}{2}(\max(s, x + x_2) + b_s(J)) + \frac{1}{2}W_k - (s + x) \geq \frac{3}{2}b_s(J) + \frac{1}{2}W_k$. All the credit of these jobs can go to J^2. The sum is at least $\frac{3}{2}b_s(J) + W_{k-1} + \frac{1}{2}w_k$. Furthermore, J^2 receives $\frac{1}{2}(x - b_s(J))$ from J, and it loses at most W_k because it is delayed by RSPT. Hence in total J^2 does not lose credit and still satisfies (6.8).

Finally, suppose there is a job in $BEFORE_s(J)$ that OPT does not complete before the final job J_k in *ARRIVE*. By Lemma 6.9 there is $\frac{3}{2}w_k$ of credit available that we can give to J^2, in addition to the $W_{k-1} + \frac{1}{2}w_k$ that it gets from the jobs in *ARRIVE*. This is sufficient for J^2 to satisfy (6.8) again. □

6.6.1 OPT runs at least three jobs before *ARRIVE*

Define $\delta(t)$ as the number of jobs OPT has completed minus the number of jobs RSPT has completed at time t. Property R4 implies that if RSPT is running a job of size x at time $t < 2x$, then $\delta(t) \leq 1$. In other words, $\delta(t) \geq 2 \Rightarrow t \geq 2x$. Thus as long as $\delta(t) \geq 2$, no jobs are ever interrupted by RSPT by Property R5.

Lemma 6.19 *If $\delta(s) \leq 1$, and a job J is completed by* RSPT*, and STATIC holds, and the invariant held at time s, then it holds at time t.*

Proof. Because of Lemma 6.15, 6.16, 6.17 and 6.18 we only need to consider the case where OPT runs $a \geq 3$ large jobs before *ARRIVE*. Since $\delta(s) \leq 1$, after time s OPT still starts $a - 2 \geq 1$ J-large job before it runs the jobs in *ARRIVE*. Therefore, RSPT completes the jobs in *ARRIVE* no later than OPT does. Moreover, $a = 3$ since the jobs in *ARRIVE* arrive before OPT completes the third J-large job. We have $K(J_i) \geq \frac{1}{2}(3x + W_i) \geq 2w_i$. We give w_i to any jobs that RSPT completes after *ARRIVE* and OPT before *ARRIVE*, since the jobs in *ARRIVE* are not interruptible and do not need any credit themselves: we have $s > x$, else $a \leq 2$. There are at most two such jobs, and their credit decreased by W because of the jobs in *ARRIVE*, using Rule C1. Therefore they get all the lost credit back from the jobs in *ARRIVE*, and again satisfy (6.8). □

Suppose $\delta(s) \geq 2$. It can only happen during the final run of a job that $\delta(s)$ increases from at most 1 to above 1, because we can apply Property R4 whenever a job is interrupted. For any maximal interval $[a, b)$ in which $\delta(s) \geq 2$ and where RSPT completes a job at time a, denote the job that it completes at time a by $J(a)$.

Lemma 6.20 *Suppose* OPT *starts its next $J(a)$-large job after time a at time s_2. Then there is a time $t \in (a, s_2 + w(J(a))]$ such that $\delta(t) \leq 1$.*

Proof. At time a, RSPT starts to run the $J(a)$-small jobs that arrived while it was running $J(a)$. Suppose $\delta(t) \geq 2$ in the entire interval $(a, s_2 + w(J(a))]$, then RSPT does not interrupt any job in this interval. Then in this interval, it certainly completes at least as many jobs as OPT starts

and completes in the interval $(a - w(J(a)), s_2]$ (it is possible that OPT decides not to run some small jobs that have arrived yet, but then it can only complete less jobs in $(a - w(J(a)), s_2]$ than RSPT does in $(a, s_2 + w(J(a))]$). Thus $\delta(s_2 + w(J(a))) \leq \delta(a) \leq 1$. □

Lemma 6.21 *Suppose $\delta(s) \geq 2$ for a job J. Then 1) $J(a)$ is J-large; 2) $s \geq 3x$; 3) at time s the total size of the smallest $\delta(s) - 1$ jobs in* RSPT*'s queue is at most* $\min(w(J(a)), a/3)$.

Proof. 1) At time $a - w(J(a))$, OPT has completed at most one job that RSPT has not completed, and such a job can only be $J(a)$-large. At time a, RSPT completes a $J(a)$-large job, namely $J(a)$ itself. OPT completes at most one $J(a)$-large job in the interval $(a - w(J(a)), a]$. Thus at time a, OPT has (still) completed at most one $J(a)$-large job more than RSPT.

By Lemma 6.20, OPT does not complete any other $J(a)$-large job within the current interval where $\delta(t) \geq 2$. On the other hand, at time s RSPT has completed all J-small jobs that have arrived, so OPT must have completed at least two J-large jobs that RSPT has not completed. Then J must be $J(a)$-small. □

2) At time s, RSPT has completed all J-small jobs that have arrived. Also it has completed $J(a)$. There are two cases. If OPT completes $J(a)$ before time s, and at least two other J-large jobs, then $s \geq w(J(a)) + 2x \geq 3x$. If OPT does not complete $J(a)$ before time s, there must be at least three other J-large jobs that OPT has completed at time s since $\delta(s) \geq 2$, and thus $s \geq 3x$. □

3) We begin by showing this holds at time a. At that time, we have that $\delta(a) - 1$ jobs that OPT has completed and RSPT has not, were started and completed by OPT in $(a - w(J(a)), a]$. Thus their total size is at most $w(J(a))$. Then the total size of the $\delta(a) - 1$ *smallest* such jobs is certainly at most $w(J(a))$.

If $a \geq 3w(J(a))$, the other bound also follows immediately. Otherwise, if OPT completes two $J(a)$-large jobs before time a, we use $a - 2w(J(a)) \leq a/3$. Finally, if it completes only one, then the first $J(a)$-small job that OPT completes in $(a - w(J(a)), a]$ must complete after time $\frac{2}{3}a$, since it does not cause an interruption. Thus the $\delta(a) - 1$ smallest jobs can start and complete in an interval of size $a/3$. (Note that if OPT completes a $J(a)$-large job in $(a - w(J(a)), a]$ in this case, then $\delta(a - w(J(a))) \leq 0$.)

Now we show that it holds later, by induction. Consider a time s for which (still) $\delta(s) \geq 2$. Denote the number of jobs that RSPT and OPT complete in $(a, s]$ by c_1 and c_2, respectively. From these c_2 jobs and the last $\delta(a) - 1$ jobs that OPT starts and completes in $(a - w(J(a)), a]$, there are then exactly $\delta(a) - 1 + c_2 - c_1 = \delta(s) - 1$ jobs that RSPT must still run. The total size of the c_1 jobs that RSPT completes in $(a, s]$ is exactly $s - a$, so the total size of these $\delta(s) - 1$ jobs is again bounded by $\min(w(J(a)), a/3)$. Then this certainly holds for the smallest $\delta(s) - 1$ jobs that RSPT must still complete. □

Lemma 6.22 *If $\delta(s) \geq 2$, and a job J is completed, and STATIC holds, and the invariant held at time s, then it holds at time t.*

Proof. Since $\delta(s) \geq 2 \Rightarrow s \geq 3x$ by part 2 of Lemma 6.21, the jobs in *ARRIVE*(I) require 0 credit because they cannot be interrupted and a larger job completes before them. By part 3 of Lemma 6.21, the $\delta(s) - 1$ smallest waiting jobs also require just 0 credit for the same reason (using $s \geq a$).

We first consider an alternative schedule, where RSPT runs the jobs in *ARRIVE*(I) not just after J, but after the $\delta(s) - 1 \geq 1$ smallest waiting jobs in RSPT's queue. (If $\delta(s) = 2$, this is only J.) The jobs in *ARRIVE*(I) then cause only small-job delay for at most one job J', and they are executed within $\frac{1}{3}s$ of their optimal starting time. Therefore each such job J_i has credit of at least $\frac{1}{2}(s + W_i) - \frac{s}{3} \geq \frac{1}{2}W_i + \frac{1}{6}s \geq \frac{1}{2}W_i + \frac{1}{2}x(I) \geq w_i$, which it can give to J' to make up for its small-job delay. By finally putting the jobs in the correct order (but keeping the credits in the same locations as usual), the total amount of credit does not decrease. This proves the lemma. □

Lemma 6.23 *If* RSPT *completes a job J, and STATIC holds, and the invariant held at time s, then it holds at time t.*

Proof. This follows from Lemmas 6.15, 6.16, 6.17, 6.18, 6.19 and 6.22. □

6.7 Interruptions, $s < x/2$

In Section 6.5, it was shown for several situations that the invariant keeps holding if an interruption occurs. There is only one case left of the situation where OPT does not complete any large jobs before *ARRIVE*, and this is the complement of Lemma 6.12: $s < x/2$. See for some examples sequences 2) and 3) in Lemma 6.1.

We will use the following Lemma. We do **not** use Global assumption 2 at time s or r_1.

Lemma 6.24 *For any input sequence σ, where some run-interval $I = (s(J), r_1]$ ends with an interruption, and where $s(J) \leq \frac{1}{2}w(J)$, and where* OPT *does not run any J-large job before ARRIVE(I), it is possible to modify the release times of some jobs so that σ can be divided into two sequences σ_1, σ_2 so that the schedule of* RSPT *for σ_1 is unchanged and the schedule for σ_2 is unchanged starting from time $s(J)$; no job in σ_2 arrives before time $s(J)$; $J \in \sigma_2$ arrives at time $s(J)$; all other J-large jobs arrive at time r_1 or later; and* OPT(σ_1) + OPT$(\sigma_2) \leq$ OPT(σ).

Proof. We divide σ into two parts: we let σ_1 contain the jobs from σ that RSPT completes before time $s(J)$, and σ_2' all the other jobs.

All jobs in σ_1 are finished by RSPT before time $s(J) \leq \frac{1}{2}w(J)$. The jobs in σ_2' either have size at least $w(J)$ or arrive after time $s(J)$ by definition of RSPT. Therefore, when processing σ, the total completion time of RSPT of the jobs in σ_1 is the same as it would have been if the jobs in σ_2' all arrived after time $s(J)$: any job in σ_2' that is running before time $s(J)$, is interrupted immediately whenever a job in σ_1 arrives, since such a job from σ_2' has size at least $w(J)$.

Moreover, OPT does not start any J-large job in σ_2' before time r_1, since OPT runs *ARRIVE*(I) before any J-large job and OPT does not complete any job in *ARRIVE*(I) before time r_1 by Property R3. Therefore, the optimal total completion time of the jobs in σ_2' is unaffected if we construct σ_2 by changing the release time of J to $s(J)$ and the release time of all other J-large jobs in σ_2' that arrive before time r_1, to r_1. Clearly, this cannot affect the optimal total completion time of the jobs in σ_1. (Note that it is possible that OPT still runs some jobs in σ_1 after time s.)

Thus RSPT(σ_1) + RSPT(σ_2) = RSPT(σ), OPT(σ_1) is the cost of σ_1 in σ, OPT(σ_2) is at most the cost of σ_2' in σ and we are done. □

Thus if there is an interruption at time r_1, and $s \leq \frac{1}{2}x$, we can consider all the jobs completed earlier as a separate job sequence and make the following assumption:

Global assumption 3 *The interrupted job was the first job in the input sequence.*

Consider such an interruption and make assumption 3. If $f = 1$, we can in fact assume **all** jobs in σ arrive at time r_1, since both OPT and RSPT start and complete all jobs in σ after time r_1. Then we are in the case where r_1 is the end of an interval in which RSPT was idle, and we apply Lemma 6.2.

In the remainder of this section, we only need to consider the case $f > 1$. The important thing about Global assumption 3 is that it implies that the first event of this sequence occurred at time s, and the job that started then still has all of its original credit. This is much more credit than could be deduced from the invariant.

Lemma 6.25 *If* RSPT *interrupts a job J at time r_1, and* OPT *runs no large jobs before ARRIVE, and $x/3 \leq s < x/2$, and $f > 1$, and STATIC holds, then the invariant holds at time r_1.*

Proof. We use Global assumption 3 and consider the credits of the jobs, assuming J arrived at time s. Again we can assume all J-large jobs besides J arrived at time r_1 (thus not using Global assumption 2 in this case). Define $D_1 = r_1 - r_f > 0$ and $D_2 = (r_f + w_f) - (r_1 + w_1)$, as before. See Table 6.10.

	1	**2**	**3**	**4**	final
J_f	$\frac{r_f+w_f}{2} - D_1$	J_1	0	$\frac{w_f-r_f-w_1}{2}$	$\frac{1}{2}w_1 + D_2$
J_1	$\frac{r_f+w_f+w_1}{2} - D_1$	J_2	$w_f - w_1$	$\frac{w_1+2r_f-w_f}{2}$	$\frac{3}{2}r_f + w_1 + D_2$
$i = 2, \ldots, f-1:$					
J_i	$\frac{r_f+w_f+W_i}{2} - D_1$	J_{i+1}	$w_f - w_i$	0	$\frac{r_f+w_f+W_{i-1}-w_i}{2} + D_2 + w_1$
$i = f+1, \ldots, k:$					
J_i	$\frac{r_f+W_i}{2} - D_1$	J_i	0	0	$\frac{r_f+W_i}{2} - D_1$
J	$\frac{r_f+W_k+x}{2} - D_1$	J	0	$-\frac{1}{2}r_f$	$\frac{W_k+x}{2} - D_1$

Table 6.10: Credit transfers in Lemma 6.25

J_1 satisfies (6.8) by Lemma 6.7, Case 1. For J_2, note that $\frac{3}{2}r_f \geq \frac{3}{2}s \geq \frac{1}{2}x \geq \frac{1}{2}w_2$, and use the same lemma. For $i = 3, \ldots, f$, we use $w_f \geq w_i$, $D_2 + w_1 = (r_f + w_f) - r_1 \geq w_f - r_1 \geq w_{i-1} - r_1$ and $D_2 + w_1 \geq 0$. This shows these jobs satisfy (6.8) by Lemma 6.7, Cases 2 and 3. For $i = f+1, \ldots, k$ we use $K(J_i) \geq \frac{1}{2}W_i - D_1 = \frac{1}{2}(w_i - w_{i-1} + W_{i-2}) + (w_{i-1} - D_1)$ and note that $w_{i-1} - D_1 \geq w_{i-1} - r_1$ and $w_{i-1} - D_1 = w_{i-1} + D_2 - w_f + w_1 \geq D_2 + w_1 \geq 0$, and we use the same Lemma. We can reason analogously for J (implying that we can indeed take $\frac{1}{2}r_f$ out of the credit of J) and for J-large jobs. □

Note that the proof of Lemma 6.25 works as long as $r_f \geq x/3$. From now on, we assume $r_f < x/3$. For this case, we use the same credit transfers as described in Table 6.10. However, in this case, this may not be enough for job J_2 to satisfy (6.8). We make one additional transfer apart from the ones mentioned in Table 6.10: we give D_2 from J_1 to J_2. By Definition 6.6, we have $D_2 = \tau_f - \tau_1$ where $\tau_f > 0$ and $\tau_1 \leq 0$. See Figure 6.3.

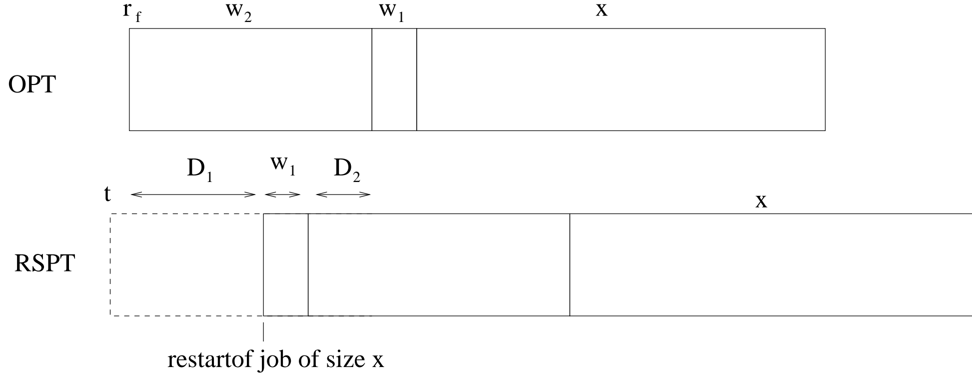


Figure 6.3: An example of a late interruption

Lemma 6.26 *Consider the jobs involved in a slow interruption as above. If J_2 does not satisfy (6.8), then*

1. $s(J_1) \geq w_1$
2. $w_f > \frac{1}{2}x$
3. $r_1 + w_1 > \frac{1}{2}x$
4. $f \in \{k-1, k\}$

Proof. After the above transfers, we have $K(J_2) = \frac{3}{2}r_f + w_1 + 2D_2$.

1. Suppose $r_1 < w_1$. Then $w_1 + D_2 = w_1 + r_f + w_f - r_1 - w_1 \geq w_f - r_1 \geq \frac{1}{2}(w_1 + w_2) - r_1$, and $w_1 + D_2 \geq 0$, so J_2 satisfies (6.8).
2. Suppose $w_f \leq \frac{1}{2}x$. Then also $w_2 \leq \frac{1}{2}x$. Moreover, since $r_f + w_f > \frac{2}{3}(s+x) \geq \frac{2}{3}x$ we have $r_f > \frac{1}{6}x$. If J_2 does not satisfy (6.8), then (using item 1) $\frac{3}{2}r_f + w_1 + 2D_2 \leq \frac{1}{2}(w_2 - w_1)$ and thus $\frac{3}{2}w_1 + 2D_2 < \frac{x}{4} - \frac{x}{4} = 0$, a contradiction.
3. Suppose $r_1 + w_1 \leq \frac{1}{2}x$. We have $D_2 \geq r_f + w_2 - (r_1 + w_1) \geq r_f + w_2 - \frac{1}{2}x$. If $w_2 \geq \frac{2}{3}x$, then $2D_2 \geq 2w_2 - x \geq \frac{1}{2}w_2$. If $w_2 < \frac{2}{3}x$, then $D_2 \geq r_f + w_f - \frac{1}{2}x \geq \frac{1}{6}x$ and $2D_2 \geq \frac{1}{2}w_2$. In both cases, we find that J_2 satisfies (6.8).
4. If $f = k-2$ or $f \leq k-4$, we can take $\frac{1}{2}w_f$ out of the credit of J and still satisfy (6.8). If $f = k-3$, we can take $\frac{1}{2}w_{f+1}$ out of $K(J)$. □

J_2 may not have enough credit to pay for its completion (it does not need to pay for interruptions of J_1, or of smaller jobs that arrive before J_2 starts). The credit to pay for the completion of J_2 will have to come from another job. We will show that we can use some of the credit of J to pay for this. To begin with, we transfer $2D_2$ of credit from J_2 to J. Then, we divide the credit of J as follows. First suppose $f = k$.

$$K_{COM}(J) = \frac{x - w_f + 4D_2}{2} \geq \frac{x + r_f - (r_1 + w_1) + 3D_2}{2} \geq \frac{1}{6}(x + r_f) + \frac{3}{2}D_2; \quad (6.15)$$

$$K_{INT}(J) = \frac{1}{2}W_{f-1} + w_f - D_1 \geq \frac{1}{2}W_{f-1} + (w_f - r_1). \quad (6.16)$$

If $f = k-1$, note that J_k cannot start before time $x > w_k$ since $r_1 + w_1 + w_f > \frac{x}{2} + \frac{x}{2} = x$ by Lemma 6.26. Hence $N_{INT}(J, r_1) \le \frac{1}{2}W_{k-2} + \frac{3}{2}w_f - r_1$. We have

$$\begin{aligned} K_{COM}(J) &= \frac{1}{2}(x - w_k) + 2D_2 \\ K_{INT}(J) &= \frac{1}{2}W_{k-1} + (w_k - D_1) \ge \frac{1}{2}W_{k-2} + (\frac{3}{2}w_k - D_1) \end{aligned}$$

so J can certainly give $\frac{1}{2}(w_k - w_f)$ to its own completion-credit, so that we again have (6.15).

We now consider the events after time r_1, using the analysis from the previous sections. Note that in those analyses, in some cases a job J' transfers its completion-credit $N_{COM}(J', s)$ to another job: this happens if J' is interrupted. However, the target job can only be a smaller job than J'. We need to keep track of the job that does not have enough completion-credit. This job will be called *red* and be denoted by J_R. Job J above, that was slowly interrupted, will be called *green* and be denoted by J_G. It satisfies (6.15) at time r_1.

Lemma 6.27 *Suppose there exists a red job J_R and a green job J_G. Until J_R completes, all jobs besides J_R and J_G satisfy (6.8), and (6.2) holds. Moreover, there will appear no further red jobs until J_R completes. J_G is the job that was slowly interrupted at time r_1, and satisfies (6.15). $K_{INT}(J_G) \ge N_{INT}(J_G, t)$ holds for all times t where an event occurs, up to and including the completion of J_R.*

Proof. We consider the events in the sequence from the first event after time r_1 until the last event before J_R completes, and use induction. At time r_1 (the base case), all the statements of the lemma hold.

Since OPT starts J_f before time r_1, it completes J_f before any job that arrives later. Since $w_R \le w_f$ by induction, we are always in the case where OPT completes at least one $J(I)$-large job before *ARRIVE*(I). This implies in particular that as long as J_R is not completed, there can occur no further slow interruptions where OPT does not run any large jobs before *ARRIVE*, so no further red jobs can appear.

Consider a later event. If it is an interruption (either of J_R, or of smaller jobs), consider the credit of the jobs in Q. As can be seen from the credit reassignments in lemmas 6.13 and 6.14, if J_R is interrupted, it is possible that another job in stead of J_R becomes red as a result. However, this can only be a job smaller than J_R. Furthermore, job J_G keeps satisfying (6.15) throughout such interruptions, since of J_R-large jobs only the amount of interrupt-credit can be affected. Also, the credit of J_G is not transferred to another job, so it is the same job that remains green. If there is an interruption of a job smaller than J_R, then J_R remains red for the same reason. In both cases, all other jobs satisfy (6.8) by the analyses in those lemmas, including the job that was J_R if another job is red now.

Now consider a completion of a job J' before J_R completes. Then J' and any smaller jobs that arrived while J' was running are all small relative to J_R and J_G. In Lemma 6.17, J_R and J_G are then among the large jobs whose credit increases by $\frac{1}{2}W(I')$, and remain red and green respectively. Their completion credit is unaffected. In Lemma 6.18 and Lemma 6.19, the same holds. In Lemma 6.22, the credit of J_R can be moved to another job (that thus becomes red), but then that is again a smaller job. □

At some point, the red job J_R will complete. By (6.7), as long as we maintain $K_{COM}(J) \geq \frac{1}{2}(x - b_s(J))$, we can take credit out of J to pay for the completion of J_R.

Lemma 6.28 *Suppose there is a red job. When it completes, credits can be transferred so that all jobs satisfy (6.8).*

Proof. Denote the set of jobs that arrive during J_R's final execution by *ARRIVE′*, and their total size by W'. Note that OPT completes $J_f \geq J_R$ before *ARRIVE′*. There are thus three cases.

Case 1. OPT completes exactly one J_R-large job before *ARRIVE′* (i.e. job J_f).

By Corollary 6.3, the jobs in *ARRIVE′* need at most $\frac{3}{2}(w'_{f'} - v'_{f'}) \leq \frac{1}{2}w_R \leq \frac{1}{2}w_f$ of credit to satisfy (6.8). We have that the credit of J increases by $\frac{1}{2}W'$.

Claim: $K(J) \geq \frac{1}{6}(x + r_f) + \frac{3}{2}D_2 + N_{INT}(J, t') + \frac{1}{2}W'$.

Proof: At the previous event, J satisfied (6.15) and $K_{INT}(J) \geq N_{INT}(J, t)$. If J_R started after time w_R, none of the jobs in *ARRIVE′* are interruptible and the claim follows. Otherwise, note that J had enough credit to pay for interruptions of J_R until time w_R, because it was green. (By Table 6.1, any job following a job J^* that is interrupted after it started before time $w(J^*)$, needs to pay for this itself until time $w(J^*)$.) This credit can now instead be used for any interruptions of jobs in *ARRIVE′* until time $2w_R$. □

Thus $\frac{1}{2}W'$ can go to the jobs in *ARRIVE′* that need it. Moreover, since $w'_{f'} - v'_{f'} \leq \frac{1}{3}w_f \leq \frac{1}{3}x$ using Corollary 6.3, we can also take $\frac{1}{2}(w'_{f'} - v'_{f'}) \leq \frac{1}{6}x$ out of the credit of J and still have $K_{COM}(J) \geq N_{COM}(J, t)$, because we take at most half the size of a job that completes before J out of J's credit, and J actually has at least this amount of credit by (6.15).

If $f' < k'$, we have $\frac{1}{2}W' \geq w'_{f'}$ and hence $\frac{3}{2}(w'_{f'} - v'_{f'}) \leq \frac{1}{2}W' + \frac{1}{2}(w'_{f'} - v'_{f'})$, which is the amount we could take from J.

If $f' = k' > 1$, we have $\frac{1}{2}W' \geq \frac{1}{2}(w'_{f'} + w'_{f'-1})$. We can give $\frac{1}{2}w'_{f'-1}$ to $J'_{f'}$ and $\frac{1}{2}w'_{f'}$ to J'_1. Also, we give $\frac{1}{2}(w'_{f'} - v'_{f'}) \leq \frac{1}{2}w'_{f'}$ from J to J'_1. It can be seen from the proof of Lemma 6.17 that this is sufficient.

If $f' = k' = 1$, then $\frac{1}{2}W' = \frac{1}{2}w'_1$. Giving this and an additional $\frac{1}{2}(w'_1 - v'_1) \leq \min(\frac{1}{6}x, \frac{1}{2}w'_1)$ from J to J'_1 is sufficient as in the proof of Lemma 6.17.

Case 2. OPT completes two J_R-large jobs before *ARRIVE′*.

If OPT runs two large jobs other than J before *ARRIVE′*, we again have that the credit of J increases by $\frac{1}{2}W'$ and we can reason as above.

Suppose OPT completes J and J_R before *ARRIVE′*. Following the proof of Lemma 6.18, we are done immediately if $s \leq x$, so suppose $s > x$. This implies the jobs in *ARRIVE′* are not interruptible, so $N_{INT}(J, t) \leq N_{INT}(J, s)$. Then $K(J'_i) = \frac{3}{2}(w_R + x) + \frac{1}{2}W'_i - (s + w_R) = \frac{1}{2}(W'_i + w_R) + \frac{3}{2}x - s$. This implies that as long as $s \leq \frac{3}{2}x$, we can give $\frac{1}{2}(w'_i + w_f) \geq w'_i$ to J from each job J'_i, which is sufficient: J receives in total at least W', which it lost because $s_t(J)$ increased.

Otherwise, note that we only need to find an extra $\frac{1}{2}w'_{k'}$ of credit to give to J, since J gets at least $W_{k'-1} + \frac{1}{2}w'_{k'}$ from the jobs in *ARRIVE′*.

Suppose $w_R \leq \frac{2}{3}x$. If $s \leq x + w_R$ then $K(J'_{k'}) \geq \frac{1}{2}(w_R + x) + \frac{1}{2}W' - w_R \geq \frac{1}{2}(x - w_R) + \frac{1}{2}w'_{k'} \geq \frac{1}{6}x + \frac{1}{2}w'_{k'} \geq \frac{3}{4}w'_{k'}$, and if $s > x + w_R$ then $K(J'_{k'}) \geq \frac{1}{2}s + \frac{1}{2}W' - w_R \geq \frac{1}{2}(x - w_R) + \frac{1}{2}w'_{k'} \geq \frac{3}{4}w'_{k'}$. We can take the last $\frac{1}{4}w'_{k'} < \frac{1}{4}w_R \leq \frac{1}{6}x$ out of the credit of J, and then all the small job-delay is paid for; J still satisfies $K_{COM}(J) \geq N_{COM}(J, t)$.

Finally, if $w_R = \frac{2}{3}x + a$ for some $a > 0$, then $D_2 = (r_f + w_f) - (r_1 + w_1) \geq s + \frac{2}{3}x + a - \frac{2}{3}(s + x) \geq a$ and we can take $\frac{1}{4}w'_{k'} + \frac{3}{4}D_2$ out of $K_{COM}(J)$ itself, since we then still have $K_{COM}(J) \geq \frac{1}{2}(x - w_f) + 2D_2 - \frac{1}{4}w_f - \frac{3}{4}D_2 \geq \frac{1}{2}(x - \frac{3}{2}(w_f - D_2)) + \frac{1}{2}D_2 \geq 0$. Here we use

$$w_f - D_2 = r_1 + w_1 - r_f \leq \frac{2}{3}(s + x) - r_f \leq \frac{2}{3}x.$$

Since we take only less than half of the size of a job that completes before J out of the credit of J, we also still have $K_{COM}(J) \geq N_{COM}(J, s)$ as before. To complete the missing credit, we can take $\frac{3}{4}(w'_{k'} - a)$ out of $K(J'_{k'})$; the calculations are similar to above.

Case 3. OPT completes three or more J_R-large jobs before *ARRIVE*$'$. Note from the proofs of Lemmas 6.19 and 6.22 that in this case, no completion credit from J_R is required to pay for any small job-delay. Hence we are done immediately. □

Chapter 7

Online scheduling of splittable tasks

In this paper, we consider the problem of distributing tasks on parallel machines, where tasks can be split into a limited amount of parts. A possible application of the splittable tasks problem exists in peer-to-peer networks [70]. In such networks large files are typically split and the parts are downloaded simultaneously from different locations, which improves the quality of service (QoS). More generally, computer systems often distribute computation between several processors. This allows the distributed system to speed up the execution of tasks. Naively it should seem that the fastest way to run a process would be to let all processors participate in the execution of a single process. However in practice this is impossible. Set-up costs and communication delays limit the amount of parallelism possible. Moreover, some processes may have limited parallelism by nature. In many cases, the best that can be done is that a process may be decomposed into a limited number of pieces each of which must be run independently on a single machine.

The definition of the model is as follows. In the sequel, we call the tasks "jobs" as is done in the standard terminology. We consider online scheduling of splittable jobs on m parallel machines. A sequence of jobs is to be scheduled on a set of machines. Unlike the basic model which assumes that each job can be executed on one machine (chosen by the algorithm), for splittable jobs, the required processing time p_j of a job j may be split in an arbitrary way into (at most) a given number of parts ℓ. Those parts become independent and may run in parallel (i.e. simultaneously) or at different times on different processors. After a decision (on the way a job is split) has been made, the scheduler is confronted by the basic scheduling problem, where each piece of job is to be assigned non-preemptively to one machine. In the on-line version, jobs are presented to the algorithm in a list, this means that each job must be assigned before the next job is revealed. Only after the process of job splitting and assignment is completed, the next job is presented to the algorithm. The goal is to minimize the makespan which is the last completion time of any part of job.

We consider two machine models. The first one is the well known model of identical machines, where all machines have the same speed (w.l.o.g. speed 1). The second case relates to systems where several processors are faster (by some multiplicative factor) than the others. In this case let s be the speed of the fast processors. The other processors have speed 1. This also contains the model where one processor is fast and all others are identical [124, 81, 37, 122]. We

call the machines of speed s *fast*, and all other machines are *regular* machines. The number of fast machines is denoted by f whereas the number of regular machines is $m-f$. The processing time of job j on a machine of speed s is p_j/s. Each machine can process only one job (or part of job) at a time, and therefore the completion time of the machine is the total processing time of all jobs assigned to it (normalized by the speed), which is also called the *load* of the machine. In the context of downloading files in a peer-to-peer network, the speeds correspond to the bandwidths for the different connections.

Our results: We first analyze a simple greedy-type algorithm that splits jobs into at most ℓ parts, while assigning them in a way that the resulting makespan is as small as possible. We improve on this algorithm by introducing a type of algorithm that always maintains a subset of $k < \ell$ machines with maximal load (while maintaining a given competitive ratio), and show that it is optimal as long as ℓ is sufficiently large in relation to $m+f$. The case $f = m-1$ is treated separately. For smaller ℓ, we give an algorithm for identical machines that uniformly improves upon our greedy algorithm. Finally, we consider the special case of four identical machines and $\ell = 2$, which is the smallest case for which we did not find an optimal solution. The algorithms assume that it is always possible to compute the value of OPT for a subsequence of jobs which already arrived. In section 7.2 we explain how to compute this value.

7.1 A greedy algorithm

In this section, we analyze a simple greedy-type algorithm that works as follows. Recall that we consider the case where there is a group of f machines of speed $s \geq 1$, and the remaining $m-f$ machines have speed 1. For each arriving job, the algorithm finds the way to schedule it on at most ℓ machines, in a way that the resulting makespan is as small as possible. This is done by assigning the job to a subset of least loaded fast machines and a subset of least loaded regular machines. To implement this algorithm, we need to consider the combination of the least loaded a regular machines with the least loaded b fast machines, for all feasible cases: $a+b \leq \ell$, $0 \leq a \leq \min\{\ell, m-f\}$ and $0 \leq b \leq \min\{\ell, f\}$. There are only $O(\ell^2)$ such combinations. If the job is split into less than ℓ parts, it means that the makespan did not change. Note that for $\ell = s = 1$, this algorithm reduces to the standard greedy algorithm for load balancing.

Consider an arbitrary subset S of ℓ machines, and denote the number of fast machines in this subset by g. Consider the time where the maximum load is achieved first. This happened after assigning a job on **exactly** ℓ machines. Denote the total processing time scheduled on the i-th machine in subset S by W_i^S ($i = 1, \ldots, \ell$). Let x be the job that achieves the maximum load (and by a slight abuse of notation, also its processing time is denoted by x). Let $W = \sum_{i=1}^{m} W_i$, i.e. the total processing time of all jobs right before the assignment of x. Let GREEDY denote the makespan of the greedy algorithm. By our assignment, we have for any subset S

$$\textsc{Greedy} \leq \frac{W_1^S + \ldots + W_\ell^S + x}{sg + \ell - g} \Rightarrow (sg + \ell - g)\textsc{Greedy} \leq W_1^S + \ldots + W_\ell^S + x.$$

There are $\binom{m}{\ell}$ such subsets, and each machine occurs in $\binom{m-1}{\ell-1}$ of them. Summing the above inequality over all $\binom{m}{\ell}$ subsets, we have that each time a fast machine occurs, it contributes s to

the left hand side; a regular machine contributes 1. Thus

$$\textsc{Greedy}\cdot\left(s\binom{m-1}{\ell-1}f+\binom{m-1}{\ell-1}(m-f)\right)\le\binom{m-1}{\ell-1}(W_1+\dots W_m)+\binom{m}{\ell}x$$

or

$$(sf+m-f)\textsc{Greedy}\le W+xm/\ell.$$

Furthermore, we have $\text{OPT}\ge\frac{W+x}{sf+m-f}$. If $f\ge\ell$ we also have $\text{OPT}\ge\frac{x}{s\ell}$, otherwise $\text{OPT}\ge\frac{x}{sf+\ell-f}$. Thus if $f\ge\ell$

$$\textsc{Greedy}\le\frac{W+x\frac{m}{\ell}}{sf+m-f}\le\text{OPT}+\frac{s\ell\cdot\text{OPT}(\frac{m}{\ell}-1)}{sf+m-f}\le\left(1+\frac{m-\ell}{sf+m-f}\cdot s\right)\text{OPT}.$$

and otherwise

$$\textsc{Greedy}\le\text{OPT}+\frac{(sf+\ell-f)\text{OPT}(\frac{m}{\ell}-1)}{sf+m-f}=\left(1+\frac{sf+\ell-f}{sf+m-f}\left(\frac{m}{\ell}-1\right)\right)\text{OPT}.$$

These ratios are decreasing in ℓ and are 1 for $\ell=m$. For $f=0$ (or equivalently $s=1$) the second ratio applies, which then becomes $2-\ell/m$. For larger f satisfying $f<\ell$, the ratio is lower.

7.2 Computing the optimal makespan

In the remainder of this paper, we assume that the value of OPT is known to the on-line algorithm. There are several options to achieve this knowledge. The algorithm of [112] can solve an offline problem exactly using time which is polynomial seeing the number of machines as constant. The drawback is that their algorithm must be exercised after every arrival of a job to find out the new value of OPT. Another and better option is simply to use the two following lower bounds on OPT: the sum of processing times of all jobs divided by the sum of speeds, and the size of the largest job divided by the sum of speeds of the ℓ fastest machines. We already used these bounds in Section 7.1.

All the proofs of upper bounds use only these bounds on OPT, and therefore the knowledge of the actual values of OPT is not required. Naturally, those bounds are not always tight as the offline problem is NP-complete already for identical machines and any constant ℓ [144]. Note that in almost all cases in this paper where we get tight bounds on the competitive ratio, the value of OPT is actually given by the maximum of the two bounds on OPT. This is always true for $\ell\ge(m+1)/2$. In these cases an optimal offline schedule (not only its cost) can be computed by the following algorithm. This algorithm works for the general case of uniformly related machines (where each machine i has some speed s_i). It is based on the sliding window algorithm from [144].

7.2.1 Offline algorithm for $\ell \geq (m+1)/2$

Calculate the maximum of the two lower bounds for OPT. We say that a job *fits* on a subset of machines if it can be placed there without any machine exceeding a load of OPT (normalized by the speed). Sort the machines by nondecreasing speeds. Consider the largest job J. Clearly it fits on the ℓ fastest machines. We consider two cases.

1. There is an index i such that J fits on machines $i, \ldots, i+\ell-1$, where all these machines except possibly the last are used completely. Assign J to those machines. We are left with $m-\ell$ empty machines and possibly a part of machine $i+\ell-1$. We have $m-\ell+1 \leq \ell$ since our assumption is that $\ell \geq (m+1)/2$. Therefore the number of machines that we can use for the remaining jobs is at most ℓ. Hence the remaining jobs can be split perfectly among these machines. Since the other machines are filled completely, they must all fit.

2. There is no such index i. In this case, J fits on machines $1, \ldots, \ell-1$ or on less machines. Note that these are the slowest machines. Therefore, the remaining jobs can be placed one by one on the machines, such that each machine has a load of OPT (except perhaps the last one). If the first part of a job is on machine i, its last part is placed on machine with index at most $i+\ell-1$. Otherwise, this job does not fit on machines $i+1, \ldots, i+\ell-1$, which means it is larger than J, a contradiction. Hence all jobs can be assigned using ℓ parts or less. Since each machine has a load of OPT apart from maybe the last machine used, all jobs fit.

These two cases show that the maximum of the two lower bounds for OPT indeed gives the true value of OPT in case $\ell \geq (m+1)/2$.

7.3 Algorithm HIGH$(k, \mathcal{R})$

An important algorithm that we work with is the following, called HIGH(k, R). It maintains the invariant that there are at least k *regular* machines with load exactly $\mathcal{R}$ times the optimal load, where $\mathcal{R}$ is the competitive ratio that we want to prove. The idea behind this algorithm is that it tries to 'fill' the regular machines, and to preserve the f fast machines for a large job that may arrive. We will use this algorithm several times in this paper, with various values of $\mathcal{R}$ and k. In all cases, we will show that a new job is never too small or too large for the invariant to be maintained.

We will use this algorithm in the context of identical machines and in the case where there are several fast machines of speed s. Recall that the identical machines case is a special case of the second case (with $s = 1$). We immediately present the more general algorithm. This algorithm also uses the sliding window technique from [144].

On arrival of a job J of size x, HIGH$(k, \mathcal{R})$ assigns the job to at most ℓ machines such that the invariant is kept. We denote the optimal makespan before the arrival of J by OPT$_1$, and after the arrival of J by OPT$_2$. We would like to sort the machines by the capacity of jobs they can accommodate. For a machine i, let L_i be its load and s' be its speed ($s' = 1$ or $s' = s$). Let b_i be the *gap* on machine i, which is the maximum load that can be placed on the machine in this

step. That is, $b_i = s'(\mathcal{R} \cdot \text{OPT}_2 - L_i)$ for $i = 1, \ldots, m$. We first sort only the regular machines in non-increasing order by their gaps. Clearly, the machines which had load $\mathcal{R}\text{OPT}_1$ have the smallest gap. We get $b_1 \geq \ldots \geq b_{m-f}$ and $b_{m-f-k+1} = \ldots = b_{m-f} = \mathcal{R}\text{OPT}_2 - \mathcal{R}\text{OPT}_1$.

Let $S_i = b_i + \ldots + b_{i+k-1}$ for $1 \leq i \leq m-f-k+1$. This is the sum of the gaps on k consecutive regular machines. The algorithm can work only under the condition that $S_{m-f-k+1} \leq x$: if x is smaller, then after assigning x there are less than k machines with load $\mathcal{R}\text{OPT}_2$. This condition will always hold for the choices of $\mathcal{R}$ and k that we analyze later. We distinguish between two cases.

1. $S_1 \geq x$. We can find a value i such that $S_i \geq x$ and $S_{i+1} \leq x$. If $S_i = x$, we can clearly assign J such that there are k regular machines with load $\mathcal{R}\text{OPT}_2$.
   Suppose $S_i > x$. Then $i \leq m - f - k$ since $S_{m-f-k+1} \leq x$. We use the machines $i, \ldots, i+k$. This is a set of $k+1$ machines. We add b_j to machine j for $j = i+1, \ldots, i+k$ and put the nonzero remainder on machine i. The remainder fits there since the job can fit on machines $i, \ldots, i+k-1$ even without machine $i+k$. Clearly we get at least k regular machines with load $\mathcal{R}\text{OPT}_2$. The assignment is feasible since $\ell \geq k+1$.

2. $S_1 < x$. Here we introduce another condition which is the following. Consider the k regular machines with the largest gaps, and among the machines that are not the k regular machines with smallest gap, choose another set of $\ell - k$ machines with largest gaps. The condition for the algorithm to succeed is that the sum of these ℓ gaps is at least the size x. The assignment of x first fills the gaps on the k least loaded regular machines, and the non-zero remainder is spread between the $\ell - k$ machines with largest gaps.

We use this algorithm several times in this paper. Each time, to show that it maintains some competitive ratio $\mathcal{R}$, we will show the following two properties.

(P1) A new job is never too large to be placed as described. That is, if we place it on the ℓ machines, k of which are the regular machines with largest gaps, and the other $\ell - k$ are the machines with the largest gaps among the others (excluding the regular machines that have maximum load before), then afterwards the load on these machines is at most $\mathcal{R}\text{OPT}_2$.

(P2) A new job is never too small for the invariant to be maintained. I.e. if we assign the job on the k machines that had load $\mathcal{R}\text{OPT}_1$, then it fits exactly in the gaps, or there is a remainder. This will show that in all cases we can make at least k machines have load $\mathcal{R}\text{OPT}_2$.

Note that for each arriving job, the new value of OPT can be computed in time $O(1)$, and the worst step in algorithm $\text{HIGH}(k, \mathcal{R})$ with regard to the time complexity is maintaining the sorted order of the regular machines, which can be done efficiently.

7.3.1 Many splits

We consider the case $\ell \geq (m+f)/2$ (since $k \leq \ell - 1$, if $f = 0$ we need $\ell \geq (m+1)/2$). Note that this leaves open the case of $f = m - 1$. This case will be considered separately in the next subsection.

We need some definitions in order to state the next Lemma. Let ℓ' be the sum of speeds of the ℓ fastest machines and let m' be the sum of all speeds. Clearly $\ell \geq f$ and so $\ell' = sf + \ell - f$ and $m' = sf + m - f$. Let $c = \ell'/m'$ and

$$\mathcal{R}_1(c) = \frac{1}{c^2 - c + 1}.$$

Note that $\mathcal{R}_1(c) = \mathcal{R}_1(1-c)$. Finally, let c_1 be the real solution to $c^3 - c^2 + 2c - 1 = 0$ ($c_1 \approx 0.56984$).

Lemma 7.1 *For $c \geq c_1$, algorithm* HIGH$(m-\ell, R_1(c))$ *maintains a competitive ratio of* $\mathcal{R}_1(c)$.

Proof Let $k = m - \ell \leq \ell - 1$. We first show that the new job is never too large to be placed as described (P1). If it is put on the ℓ machines which are all machines that did not have maximum load before the arrival of J, then the other $k = m - \ell$ regular machines have load $\mathcal{R}_1(c)\text{OPT}_1$ because of the invariant (they were the machines with highest load). Thus we need to show that $\ell'\mathcal{R}_1(c)\text{OPT}_2 + k\mathcal{R}_1(c)\text{OPT}_1 \geq W + x$ where W is the total load of all the jobs before J arrived.

We have $\text{OPT}_1 \geq W/m'$, $\text{OPT}_2 \geq (W+x)/m'$ and $\text{OPT}_2 \geq x/\ell'$. Therefore

$$\text{OPT}_2 \geq \alpha\frac{W+x}{m'} + (1-\alpha)\frac{x}{\ell'} \qquad \text{for any } 0 \leq \alpha \leq 1 \tag{7.1}$$

Taking $\alpha = \ell'/m'$, we get $k\text{OPT}_1 + \ell'\text{OPT}_2 \geq kW/m' + \ell'\alpha(W+x)/m' + \ell'(1-\alpha)x/\ell' = (W+x)(\alpha\ell'/m' + 1 - \alpha) = (W+x)(1 - \ell'/m' + \ell'^2/m'^2) = \frac{W+x}{\mathcal{R}_1(c)}$, as needed.

Second, we show that J is always large enough such that we can again make k regular machines have load $\mathcal{R}_1(c)\text{OPT}_2$ (P2). That is, $x \geq k\mathcal{R}_1(c)(\text{OPT}_2 - \text{OPT}_1)$. There are three possibilities for OPT_2: it is either x/ℓ', $(W+x)/m'$ or y/ℓ', where y is the processing time of some old job.

If $\text{OPT}_2 = y/\ell'$ we are done, since then $\text{OPT}_1 = y/\ell'$ as well. Otherwise, we use that $\text{OPT}_1 \geq W/m'$. Thus $\text{OPT}_2 - \text{OPT}_1 \leq \max(x/\ell', x/m') = x/\ell'$. We need to show that $k\mathcal{R}_1(c)x/\ell' \leq x$ or $k\mathcal{R}_1(c) \leq \ell'$. This holds if $c^3 - c^2 + 2c - 1 \geq 0$, which holds for $c \geq c_1$. This completes the proof of the upper bound of HIGH$(m-\ell, \mathcal{R}_1(c))$. □

Lemma 7.2 *No algorithm for the scheduling of ℓ-splittable jobs on a system of f fast machines of speed s and $m - f$ regular machines has a better competitive ratio than $\mathcal{R}_1(c)$.*

Proof The values m' and ℓ' are defined as above. Thus $m' = sf + m - f$. Furthermore, ℓ' is the sum of speeds of the ℓ fastest machines, so $\ell' = sf + \ell - f$ if $\ell \geq f$, $\ell' = s\ell$ otherwise. The lower bound consists of very small jobs of total size $m' = sf + m - f$, followed by a single job of size $W - m'$, where W will be determined later. The optimal offline makespan after the small jobs is $\text{OPT}_1 = 1$, and after the large job it is $\text{OPT}_2 = W/m'$.

Consider an online algorithm $\mathcal{A}$. After the small jobs have arrived, the algorithm "knows" it has to keep room for another single job. Therefore it can load the $m - \ell$ machines it is not going to use for that job with the maximum load $\mathcal{R}\text{OPT}_1$ (if it puts more on some machine, the final job does not arrive). There are many cases according to how many fast machines it loads. Let k_1

be the number of fully loaded regular machines and $k_2 = m - \ell - k_1$ the number of fully loaded fast machines.

If $\mathcal{A}$ maintains a competitive ratio of $\mathcal{R}$, we must have

$$W \leq \mathcal{R}\textsc{Opt}_1(k_1 + sk_2) + \mathcal{R}\textsc{Opt}_2((m - f - k_1) + s(f - k_2)). \tag{7.2}$$

This implies

$$\mathcal{R} \geq \frac{W}{m - \ell - k_2 + sk_2 + \textsc{Opt}_2(k_2 + \ell - f + sf - sk_2)}. \tag{7.3}$$

We can see that this number is minimized by minimizing k_2, since the coefficient of k_2 in the denominator is $(\textsc{Opt}_2 - 1)(1 - s) < 0$. Therefore the lower bound is obtained by taking $k_2 = 0$ if $\ell \geq f$, and $k_2 = f - \ell$ otherwise. We choose W such that $W - m' = m'\ell'/(m' - \ell')$. We rewrite (7.2) to get $W \leq (m' - \ell')\mathcal{R}\textsc{Opt}_1 + \ell'\mathcal{R}\textsc{Opt}_2$. Then since $\textsc{Opt}_1 = 1$ and since from $W = (m')^2/(m' - \ell')$ follows $\textsc{Opt}_2 = m'/(m' - \ell')$, we get $\mathcal{R} \geq \frac{(m')^2}{(m'-\ell')^2 + m'\ell'} = \frac{(m')^2}{(m')^2 - m'\ell' + (\ell')^2} = \mathcal{R}_1(c)$. □

These two lemmas imply the following theorem.

Theorem 7.1 *For $\ell'/m' \geq c_1$ and $\ell \geq \frac{m}{2} + \frac{1}{2}\max(f, 1)$ (i.e. $f \neq m - 1$), the algorithm $\textsc{High}(m - \ell, \mathcal{R}_1(\ell'/m'))$ is well-defined and optimal.*

7.3.2 The case of $f = m - 1$ fast machines

For completeness, in this section we consider the case $f = m - 1$. We give tight bounds for many cases, including the case of $m - 1$ parts, i.e. each job may run on all machines but one. Clearly we already solved the cases $f = 0, \ldots, m - 2$ and $f = m$ (this is the same case as $f = 0$) for large enough ℓ. The solution of the case $f = m - 1$ is very different from the other cases. First the algorithm is not the same for all values of s. For small s, for the first time we use an invariant on the fast machines. For large s, for the first time we do not use all the machines. Again we use m' as the sum of all speeds, i.e. $m' = (m - 1)s + 1$, and ℓ' as the sum of speeds of the ℓ fastest machines, i.e. $\ell' = s\ell$. We introduce a new notation k' which is the sum of speeds of the machines that are kept at maximum load. This value is determined by the algorithm.

For large s, we use an algorithm which never uses the regular machine. For the case $\ell = m - 1$ it is a simple greedy algorithm that splits each job in a way that it keeps the load balanced on all fast machines. This gives the algorithm the ratio $1 + \frac{1}{s(m-1)}$ (easily proved by area considerations). For $\ell < m - 1$ the algorithm ignores the regular machine, and uses $\textsc{High}(m - 1 - \ell, \mathcal{R}_{21})$ on $m - 1$ fast machines only, where $\mathcal{R}_{21}$ is defined as a function of m' and ℓ' (which are functions of m, ℓ and s):

$$\mathcal{R}_{21} = \frac{(m')^2}{(m')^2 - (m' - \ell')(\ell' + 1)} = \frac{(m')^2}{(m')^2 - m' - k'\ell'}.$$

We have $k' = sk = s(m - \ell - 1)$. The algorithm keeps $k = m - \ell - 1$ fast machines with load $\mathcal{R}_{21}\textsc{Opt}$. Since k must be smaller than ℓ, we require $\ell \geq m/2$.

On arrival of a job, let OPT_1 and OPT_2 be the optimal offline makespan before and after the arrival of the new job, respectively. The algorithm is the same as before but the properties are slightly different. We need to show that the following two properties hold:

(P1) $x \geq k'\mathcal{R}(\text{OPT}_2 - \text{OPT}_1)$.

(P2) The gaps on the ℓ least loaded fast machines can contain x.

The second property can be reformulated as

$$\ell'\mathcal{R}\text{OPT}_2 + k'\mathcal{R}\text{OPT}_1 \geq W + x$$

where W is the total processing time of jobs which arrived before the job of processing time x. This follows from $\ell + k = m - 1$. Regarding (P1), similarly to before, we can bound the difference of the optimal offline costs by $\text{OPT}_2 - \text{OPT}_1 \leq x/\ell'$. This gives the condition $\mathcal{R}_{21} \leq \ell'/k'$.

To show (P2) we again use the bounds $\text{OPT}_1 \geq \frac{W}{m'}$ and (7.1). We need to show

$$\frac{k'W}{m'} + \ell'\left(\alpha\frac{W+x}{m'} + (1-\alpha)\frac{x}{\ell'}\right) \geq \frac{W+x}{\mathcal{R}}.$$

Taking $1 - \alpha = \frac{k'}{m'}$, we get that this condition is satisfied for $\mathcal{R} = R_{21}$.

For small s, we use a variation on previous algorithms. The algorithm keeps $k = m - \ell$ *fast* machines with load $\mathcal{R}\text{OPT}$, where

$$\mathcal{R}_{22} = \frac{m'^2}{m'^2 - (m' + s - 1 - \ell')\ell'} = \frac{(m')^2}{(m')^2 - k'\ell'}. \tag{7.4}$$

The value we use for k' is $k' = s(m - l)$. The algorithm is defined as $\text{HIGH}(m - \ell, \mathcal{R}_{22})$, except that the roles of the fast machines and the regular machine have been reversed. In other words, we use the gaps on *fast* machines to fit the job, and if it needs more room we use at most $m - k - 1$ fast machines and the regular machine as well.

On arrival of a job, let OPT_1 and OPT_2 be the optimal offline makespan before and after the arrival of the new job, respectively. We again need the following two properties to hold:

(P1) $x \geq k'\mathcal{R}(\text{OPT}_2 - \text{OPT}_1)$.

(P2) The gaps on the $m - k$ other machines (that do not maintain the invariant) can contain x.

$$(m' - k')\mathcal{R}\text{OPT}_2 + k'\mathcal{R}\text{OPT}_1 \geq W + x.$$

(P1) again translates into $\mathcal{R}_{22} \leq \ell'/k'$. To show (P2) we again use the bounds $\text{OPT}_1 \geq \frac{W}{m'}$ and (7.1). We need to show

$$\frac{k'W}{m'} + (m' - k')\left(\alpha\frac{W+x}{m'} + (1-\alpha)\frac{x}{\ell'}\right) \geq \frac{W+x}{\mathcal{R}}.$$

Taking $1 - \alpha = \frac{k'\ell'}{m'(m'-k')}$, we get that this condition is satisfied for $\mathcal{R} = R_{22}$.

We now give a lower bound that proves that these bounds are tight. The lower bound is actually more general, and holds for all values of ℓ and s.

Lemma 7.3 *For $f = m-1$, any online algorithm has competitive ratio at least* $\min(\mathcal{R}_{21}, \mathcal{R}_{22})$.

Proof We define a sequence of jobs with the following processing times: $P_1 = 1$, $P_j = \frac{\ell'}{m'-\ell'}\sum_{i=1}^{j-1} P_i$. Let OPT_j be the optimal offline cost on the subsequence of the first j jobs. Then we see that for $j \geq 3$ we have

$$\text{OPT}_j = \frac{1}{m'-\ell'}\sum_{i=1}^{j-1} P_i = \frac{P_j}{\ell'} \quad \text{and} \quad P_j = \frac{m'}{m'-\ell'}P_{j-1}.$$

Consider the behavior of the on-line algorithm starting from the **third job**.

If the algorithm never splits a job using the regular machine, we need to consider two cases. If $\ell = m-1$, the competitive ratio tends to the ratio $1 + \frac{1}{s(m-1)}$ of the greedy algorithm that does not use the regular machine. The second case $\ell \leq m-2$ is slightly more difficult. Only the first two jobs might be scheduled on the regular machine. Consider job P_j. If $\mathcal{A}$ maintains a competitive ratio of $\mathcal{R}$ until this point, then on each of the fast machines that it does not use for job j it has placed a load of at most $sR\text{OPT}_{j-1}$, and we find

$$\frac{\sum_{i=3}^{j} P_i - (m-\ell-1)sR\text{OPT}_{j-1}}{\ell'} \leq \mathcal{R}\text{OPT}_j$$

which implies that $\mathcal{R}(\ell'\text{OPT}_j + s(m-\ell-1)\text{OPT}_{j-1}) + P_1 + P_2 \geq \sum_{i=1}^{j} P_i$. We use $\sum_{i=1}^{j} P_i = P_j + \sum_{i=1}^{j-1} P_i = P_j(1 + \frac{m'-\ell'}{\ell'}) = \frac{m'}{\ell'}P_j$ to rewrite this condition in terms of P_j, and divide by P_j. For large enough j we can neglect P_1 and P_2 and find

$$\mathcal{R}\left(1 + \frac{s(m-\ell-1)(m'-\ell')}{m'\ell'}\right) \geq \frac{m'}{\ell'}.$$

This gives $\mathcal{R} \geq \mathcal{R}_{21}$.

Otherwise (some job uses the regular machine), let j be the index of the first job for which a part is assigned to the regular machine. If $\mathcal{A}$ maintains a competitive ratio of $\mathcal{R}$ until this point, then on the machines that it does not use for job j (which are all fast) it has placed at most $sR\text{OPT}_{j-1}$, and we find

$$\frac{\sum_{i=1}^{j} P_i - s(m-\ell)R\text{OPT}_{j-1}}{s(\ell-1)+1} \leq \mathcal{R}\text{OPT}_j$$

which implies that $\mathcal{R}(\text{OPT}_j(s(\ell-1)+1)+s(m-\ell)\text{OPT}_{j-1}) \geq \sum_{i=1}^{j} P_i$. We use $\sum_{i=1}^{j} P_i = \frac{m'}{\ell'}P_j$ to rewrite this condition in terms of P_j, and divide by P_j to find

$$\mathcal{R}\left(\frac{s\ell - s + 1}{\ell'} + \frac{s(m-\ell)(m'-\ell')}{\ell' m'}\right) \geq \frac{m'}{\ell'}$$

which leads to $\mathcal{R} \geq \mathcal{R}_{22}$. □

We summarize our results in the following Theorem.

Let $s_1 = (m - 1 + \sqrt{m^2 - 2m + 1 + 4\ell})/(2\ell)$.

Theorem 7.2 *For the case of* $m-1$ *fast machines of speed* s. *If* $s \geq s_1$, *and* ($m/2 \leq \ell \leq m-2$ *and* $\mathcal{R}_{21} \leq \ell'/(m'-\ell'-1)$) *or* $\ell = m-1$, *then the optimal competitive ratio of any online algorithm is* $\mathcal{R}_{21}$. *If* $s \leq s_1$, $\ell > m/2$ *and* $\mathcal{R}_{22} \leq \ell'/(m'-\ell'+s-1)$, *then the optimal competitive ratio of any online algorithm is* $\mathcal{R}_{22}$.

Corollary 7.1 *For* $f = \ell = m-1$, *the optimal competitive ratio is* $\min(R_{21}, \mathcal{R}_{22})$.

Proof For small s, if $\ell = m-1$ then the value of $\mathcal{R}_{21}$ is defined properly to be $1+\frac{1}{s(m-1)}$, attained by the greedy algorithm that only uses fast machines. This ratio is thus tight.

For large s, if $\ell = m-1$ then the first property to be checked leads to the condition $s\mathcal{R}(\text{OPT}_2 - \text{OPT}_1) \leq x$. Similarly to before, we can bound the difference of the optimal offline costs by $\text{OPT}_2 - \text{OPT}_1 \leq x/(sm-s)$. Using (7.4), this leads to the condition $s^2(m-1)^2 \leq (m-2)(sm-s+1)^2$. This is true since $s(m-1) < sm-s+1$ and $m \geq 3$. Thus the condition on the ratio in Theorem 7.2 is satisfied as well as the condition on ℓ. □

7.3.3 Few splits on identical machines

Following Theorem 7.1, we now consider the case $c < c_1 \approx 0.56984$. Let

$$\mathcal{R}_3(c) = \frac{1}{2}\left(c^2 - c + 2 - (c-1)\sqrt{c^2+4}\right).$$

We examine algorithm $\text{HIGH}(\ell/\mathcal{R}_3(c), \mathcal{R}_3(c))$, i.e. $k = \ell/\mathcal{R}_3(c)$, and verify that it maintains a competitive ratio of $\mathcal{R}_3(c)$. Condition (P2) is immediately satisfied, since the only relevant case is $\text{OPT}_2 - \text{OPT}_1 \leq x/\ell$, which leads to the constraint $kR_3(c) \leq \ell$ as in the previous subsection. Moreover, we have that $k+\ell \leq m$ for all $c \leq c_1$, since $c/R_3(c) + c \leq 1$ for $c < c_1$.

Suppose a new job is placed on the ℓ machines with lowest load. By the invariant and since $k+\ell \leq m$, there are k machines with load $\mathcal{R}_3(c)\text{OPT}_1$. Denote the total load on the remaining machines (not the k old machines or the ℓ machines that were just used) by V. Then

$$V \geq (W - k\mathcal{R}_3(c)\text{OPT}_1) \cdot \frac{m-k-\ell}{m-k}$$

since these machines were not the least loaded machines before the new job arrived.

Thus we need to check that

$$k\mathcal{R}_3(c) \cdot \text{OPT}_1 + \ell\mathcal{R}_3(c) \cdot \text{OPT}_2 + V \geq W + x$$

or

$$k\mathcal{R}_3(c) \cdot \text{OPT}_1 \cdot \frac{\ell}{m-k} + \ell\mathcal{R}_3(c) \cdot \text{OPT}_2 \geq W \cdot \frac{\ell}{m-k} + x.$$

As before, we use that $\text{OPT}_1 \geq W/m$ and $\text{OPT}_2 \geq \alpha\frac{W+x}{m} + (1-\alpha)\frac{x}{\ell}$ for any $0 \leq \alpha \leq 1$. We take $\alpha = \frac{m-k}{2m-k-\ell} \leq \frac{m-k}{m} \in [0,1]$.

We find

$$\frac{k\ell\text{OPT}_1}{m-k} + \ell\text{OPT}_2 \geq \left(\frac{k\ell}{m-k} + \ell\alpha\right)\frac{W}{m} + \left(\ell \cdot \frac{\alpha}{m} + 1 - \alpha\right)x \geq \frac{\frac{W\ell}{m-k} + x}{\mathcal{R}_3(c)},$$

since $\mathcal{R}_3(c)$ satisfies $\mathcal{R}_3(c) = \frac{2m-k-cm}{m-kc}$ (using $k = \ell/\mathcal{R}_3(c) = cm/\mathcal{R}_3(c)$).

Theorem 7.3 *For $\ell/m < c_1$, the algorithm* $\text{HIGH}(\ell/\mathcal{R}_3(c), \mathcal{R}_3(c))$ *maintains a competitive ratio of $\mathcal{R}_3(c)$, where $c = \ell/m$.*

We now show a lower bound for this case. This lower bound uses a technique originally introduced by Sgall [142, 143].

Theorem 7.4 *For m divisible by ℓ, the competitive ratio of any randomized (or deterministic) algorithm is at least $\frac{1}{1-(1-\frac{\ell}{m})^{m/\ell}}$. This gives a general lower bound of $\mathcal{R}_4(c) = \left(1 - \left(\frac{c-1}{c}\right)^c\right)^{-1}$ for $c = \ell/m$.*

Proof Fix a sequence of random bits to be used by the algorithm. Start with $(m-\ell)/\ell$ jobs of size ℓ. Then define $\mu = m/(m-\ell)$ and give jobs J_i of size $\ell\mu^{i-1}$ for $i = 1, \ldots, m/\ell$.

Since $\mu - 1 = \ell/(m-\ell)$, we have $\sum_{i=1}^{m/\ell} \ell\mu^{i-1} = \ell\frac{\mu^{m/\ell}-1}{\mu-1} = (m-\ell)(\mu^{m/\ell}-1)$. Therefore the total size of all the jobs is $W = m - \ell + (m-\ell)(\mu^{m/\ell}-1) = (m-\ell)\mu^{m/\ell} = m\mu^{m/\ell-1}$. After job J_i has arrived we have $\text{OPT}_i = \mu^{i-1}$. So $\sum_{i=1}^{m/\ell} \text{OPT}_i = (\mu^{m/\ell}-1)/(\mu-1)$.

For $1 \le i \le m$, let L_i be the load of machine i at the end of the sequence after sorting the machines by non-increasing load. Removing any $i-1$ jobs still leaves a machine with load of at least $L_{\ell i+1}$. Therefore $\mathcal{A}(J_m) = L_1$, $\mathcal{A}(J_{m-1}) \ge L_{\ell+1}$ and in general $\mathcal{A}(J_i) \ge L_{\ell(m-i)+1} \ge \frac{1}{\ell}\sum_{j=1}^{\ell} L_{\ell(m-i)+j}$, so $\sum \mathcal{A}(J_i) \ge W/\ell$.

It follows that

$$\begin{aligned}\mathcal{R} &\ge \frac{W/\ell}{\sum_{i=1}^{m/\ell}\text{OPT}_i} \ge \frac{m\mu^{m/\ell-1}(\mu-1)/\ell}{\mu^{m/\ell}-1} = \frac{\mu^{m/\ell}}{\mu^{m/\ell}-1} \\ &= \frac{1}{1-\left(\frac{m}{m-\ell}\right)^{-m/\ell}} = \frac{1}{1-\left(1-\frac{\ell}{m}\right)^{m/\ell}}.\end{aligned}$$

The value of the lower bound tends to $e/(e-1)$ for $m/\ell \to \infty$, for instance when ℓ is constant and m grows without bound. For $m = c\ell$ we find a lower bound of

$$\mathcal{R}_4(c) = \left(1 - \left(\frac{c-1}{c}\right)^c\right)^{-1},$$

independent of m. □

We give an overview of the various upper and lower bounds in Figure 1.

7.4 A special case: four machines, two parts

Already for this sub-problem it is nontrivial to give an optimal algorithm. Surprisingly, in this case the lower bound from Theorem 7.4 is not tight. This hints that for the cases where we do not give matching upper bounds, it is likely that the lower bounds are simply not the best possible.

For the case of three parts the previous section gives an algorithm of competitive ratio $16/13 \approx 1.23$. For two parts, we use the algorithm $\text{HIGH}(1, 10/7)$ which maintains the invariant that at least one machine has load exactly $\frac{10}{7}\text{OPT}$. Note that our greedy algorithm maintains only a competitive ratio of $1 + \frac{\ell}{m}(\frac{m}{\ell} - 1) = 3/2$.

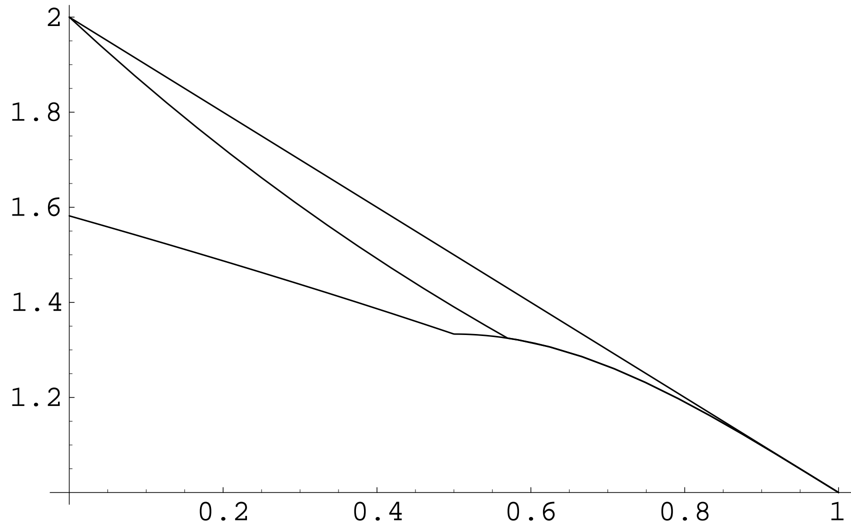


Figure 7.1: Upper and lower bounds for identical machines. The horizontal axis is ℓ/m, the vertical axis is the competitive ratio. The top line is the greedy algorithm, the middle line is our best upper bound and the lower line is our best lower bound. For $c \leq 1/2$, this lower bound also holds for randomized algorithms.

Theorem 7.5 *For four machines and 2-splittable jobs, the algorithm* HIGH$(1, 10/7)$ *maintains a competitive ratio of* $10/7 \approx 1.428$.

Proof The proof proceeds similarly to before.

First, we show that a new job, J, is not too large (P1). Suppose it is placed on the two lowest machines. Then the other machines have the loads $\frac{10}{7}\text{OPT}_1$ (because of the invariant) and $\beta \geq (W - \frac{10}{7}\text{OPT}_1)/3$ (because it was the second highest machine before J arrived). The total load on all the machines must be bounded by $\frac{10}{7}\text{OPT}_1 + \frac{20}{7}\text{OPT}_2 + \beta \geq \frac{10}{7}(\frac{2}{3}\text{OPT}_1 + 2\text{OPT}_2) + W/3$.

Recall that $\text{OPT}_1 \geq W/4$ and $\text{OPT}_1 \geq \max((W + x)/4, x/2)$; then, using (7.1), we have

$$\frac{\text{OPT}_1}{3} + \text{OPT}_2 \geq \frac{W}{12} + \frac{\alpha W}{4} + \frac{\alpha x}{4} + (1 - \alpha)\frac{x}{2} = \frac{7}{60}(2W + 3x)$$

by taking $\alpha = 3/5$. Therefore $\frac{10}{7}\text{OPT}_1 + \frac{20}{7}\text{OPT}_2 + \beta \geq W + x$, as needed.

Second, we show that a new job is always large enough so that the new maximum load is $10/7$ times the optimal load (P2). We have $\text{OPT}_2 - \text{OPT}_1 \leq x/2$, and $\frac{10}{7}\frac{x}{2} \leq x$. □

Lemma 7.4 *Any on-line algorithm for minimizing the makespan of 2-splittable jobs on four parallel machines has a competitive ratio of at least* $\mathcal{R}_4 = (47 - \sqrt{129})/26 \approx 1.37085$.

Proof Suppose $\mathcal{A}$ maintains a competitive ratio of R. Two jobs of size 2 arrive. $\text{OPT} = 1$ (already after the first job). We number the machines from 1 to 4, and denote the loads of the machines by $M_1 \geq M_2 \geq M_3 \geq M_4$. If $\mathcal{A}$ puts the first two jobs on two or fewer machines, we are done immediately. This leaves us with two cases. We use $\mathcal{A}$ to also denote the makespan of $\mathcal{A}$.

Case 1. $\mathcal{A}$ puts the first two jobs on 3 machines. Then $M_4 = 0$, $M_1 \leq \mathcal{R}_4$, $M_3 \geq 4 - 2\mathcal{R}_4$, $M_2 + M_3 \geq 4 - \mathcal{R}_4$ and therefore $M_2 \geq (4 - \mathcal{R}_4)/2 = 2 - \mathcal{R}_4/2$.

A job x of size 2 arrives. If $\mathcal{A}$ puts no part of x on machine 4, we are done since $M_3 + 1 \geq 5 - 2\mathcal{R}_4 > 3\mathcal{R}_4/2$ (we have OPT $= 3/2$).

So $\mathcal{A}$ must put a part of x on machine 4. Finally, a job of size 6 will arrive. The best thing $\mathcal{A}$ can do is to put it on the two machines with lowest load (after x has been assigned). Which machines are these?

Case	Lowest load is on	and is at least
1a	2 and 3, 1 and 3 or 1 and 2	$4 - \mathcal{R}_4$
1b	2 and 4	$8 - 4\mathcal{R}_4$
1c	3 and 4	$8 - 4\mathcal{R}_4$

This covers the cases, since if part of x is put on 4, either machine 2 or machine 3 receives nothing and remains lower than machine 1. We now prove the entries in the last column.

(1a) Suppose machines 2 and 3 are the lowest. Already before assigning x we had $M_2 + M_3 \geq 4 - \mathcal{R}_4$. Now suppose machines 1 and 3 are the lowest. Clearly $M_1 + M_3 \geq M_2 + M_3 \geq 4 - \mathcal{R}_4$. Finally, if machines 1 and 2 are the lowest then $M_1 + M_2 \geq M_3 + M_2 \geq 4 - \mathcal{R}_4$.

(1b) It must be that x goes to machines 3 and 4. $\mathcal{A}$ should put as little as possible on 4 in order to minimize the load on the two lowest machines after this (2 and 4). $\mathcal{A}$ can put at most $3\mathcal{R}_4/2 - (4 - 2\mathcal{R}_4) = 7\mathcal{R}_4/2 - 4$ on machine 3 and thus puts at least $2 - (7\mathcal{R}_4/2 - 4) = 6 - 7\mathcal{R}_4/2$ on machine 4. After this, the load of the two lowest machines (2 and 4) is at least $2 - \mathcal{R}_4/2 + 6 - 7\mathcal{R}_4/2 = 8 - 4\mathcal{R}_4$.

(1c) Again $\mathcal{A}$ should put as little as possible on 4 in order to minimize the load on the two lowest machines after this (3 and 4). It can put at most $3\mathcal{R}_4/2 - (2 - \mathcal{R}_4/2) = 2\mathcal{R}_4 - 2$ on machine 1 or 2 and must therefore put at least $2 - (2\mathcal{R}_4 - 2) = 4 - 2\mathcal{R}_4$ on machine 4. After this, the load of the two lowest machines (3 and 4) is at least $8 - 4\mathcal{R}_4$.

This concludes the discussion of the subcases. We find that after assigning x, the load on the two lowest machines is at least $\min(4 - \mathcal{R}_4, 8 - 4\mathcal{R}_4) = 8 - 4\mathcal{R}_4$ since $\mathcal{R}_4 > 4/3$. Finally the job of size 6 arrives, now OPT $= 3$ and $\mathcal{A} \geq (8 - 4\mathcal{R}_4 + 6)/2 > 3\mathcal{R}_4$.

Case 2. $\mathcal{A}$ puts the first two jobs on 4 machines, each machine has one part of one job. Then $M_2 + M_3 = M_1 + M_4 = 2$ and

$$M_1 \leq \mathcal{R}_4.$$

It is possible that a job of size 4 arrives. Then OPT $= 2$ and $\mathcal{A}$ must be able to place it such that $\mathcal{A} \leq 2\mathcal{R}_4$. Therefore we must have $(M_3 + M_4 + 4)/2 \leq 2\mathcal{R}_4$ or

$$M_3 + M_4 \leq 4(\mathcal{R}_4 - 1).$$

Together these equations give

$$M_1 + M_2 \geq 8 - 4\mathcal{R}_4, \qquad M_2 \geq 8 - 5\mathcal{R}_4 \qquad \text{and} \qquad M_4 = 2 - M_1 \geq 2 - \mathcal{R}_4.$$

Thus, if these inequalities do not hold after the first two jobs arrive, a job of size 4 arrives and we are done. Otherwise, we let a job of size x ($x \leq 1$) arrive where x will be determined later. Then OPT $= 1 + x/4$. After this a final job of size $y = x + 4$ will arrive. We have a similar division into cases as in Case 1.

Case	Lowest load is on	and is at least
2a	2 and 3, 1 and 3 or 1 and 2	2
2b	(1 or 2) and 4	$10 - 6\mathcal{R}_4$
2c	3 and 4	$10 - 6\mathcal{R}_4$

(2a) We have $M_2 + M_3 = 2$, the rest is as in Case 1a.

(2b) We have $M_2 + M_4 \geq 10 - 6\mathcal{R}_4$, so also $M_1 + M_4 \geq 10 - 6\mathcal{R}_4$.

(2c) We are left with the case where machines 3 and 4 are the lowest. We will choose x so large that it cannot be assigned to machines 1 and 2 only: $M_1 + M_2 + x > 2\mathcal{R}_4(1 + x/4)$, in other words $x > (12\mathcal{R}_4 - 16)/(2 - \mathcal{R}_4)$.

Thus some part of x is assigned to machine 3 or 4. $\mathcal{A}$ will use machines 3 and 4 for the last job, so it is best to put as much of x as possible on 1 or 2. WLOG this part is put on machine 2 since $M_2 \leq M_1$. Denote the part of x that is assigned to machine i by x_i. We have $x_2 \leq (1 + x/4)\mathcal{R}_4 - M_2$ and

$$M_3 + x_3 = 2 - M_2 + x - x_2 \geq 2 - M_2 + x(1 - \mathcal{R}_4/4) - \mathcal{R}_4 + M_2 = 2 - \mathcal{R}_4 + x(1 - \mathcal{R}_4/4).$$

Therefore $M_3 + M_4 + x_3 \geq 4 - 2\mathcal{R}_4 + x(1 - \mathcal{R}_4/4)$.

We take x such that $10 - 6\mathcal{R}_4 = 4 - 2\mathcal{R}_4 + x(1 - \mathcal{R}_4/4)$, in other words $x = (24 - 16\mathcal{R}_4)/(4 - \mathcal{R}_4) = (16\sqrt{129} - 128)/(\sqrt{129} + 57) \approx 0.7859$. Note that $x > (12\mathcal{R}_4 - 16)/(2 - \mathcal{R}_4)$, as needed.

This concludes the discussion of the subcases. We find that the load of the two lowest machines is at least $10 - 6\mathcal{R}_4$ after assigning job x, independently of $\mathcal{A}$'s decision. (Note $10 - 6\mathcal{R}_4 < 2$ for $\mathcal{R}_4 > 4/3$.)

After the last job arrives, OPT $= y/2$. The best thing that $\mathcal{A}$ can do is to put y on the two machines with lowest load. Its final load is thus at least $(10 - 6\mathcal{R}_4 + y)/2$. The competitive ratio is $(10 - 6\mathcal{R}_4 + 4 + x)/(4 + x) = \mathcal{R}_4$.

□

7.5 Conclusion

This chapter considered the classical load balancing model in the context of parallelizable tasks. We designed and analyzed several algorithms, and showed tight bounds for many cases. As for open problems, there is a large amount of work done on various multiple machines scheduling and load balancing problems. Many of those on-line (and offline) problems are of interest to be studied for scenarios where parallelization is allowed.

For the special case of four machines and two parts, which is the smallest case for which we do not have a tight solution, we show a lower bound of 1.37085 and an upper bound of $10/7 \approx 1.428$. This is a better lower bound than Lemma 7.2, hinting that in areas where our bounds are not tight, the lower bound can be improved.

Chapter 8

Speed scaling of tasks with precedence constraints

8.1 Motivation

Power is now widely recognized as a first-class design constraint for modern computing devices. This is particularly critical for mobile devices, such as laptops, that rely on batteries for energy. While the power consumption of devices has been growing exponentially, battery capacities have been growing at a (modest) linear rate. One common technique for managing power is speed/voltage/power scaling. For example, current microprocessors from AMD, Intel and Transmeta allow the speed of the microprocessor to be set dynamically. The motivation for speed scaling as an energy saving technique is that, as the speed to power function $P(s)$ in all devices is strictly convex, less aggregate energy is used if a task is run at a slower speed. The application of speed scaling requires a policy/algorithm to determine the speed of the processor at each point in time. The processor speed should be adjusted so that the energy/power used is in some sense justifiable by the improvement in performance attained by running at this speed.

In this paper, we consider the problem of speed scaling to conserve energy in a multiprocessor setting where there are precedence constraints between tasks, and where the performance measure is the makespan, the time when the last task finishes. We will denote this problem by $Sm \mid prec, energy \mid C_{max}$. Without speed scaling, this problem is denoted by $Pm \mid prec \mid C_{max}$ in the standard three field scheduling notation [85]. Here m is the number of processors. This is a classic scheduling problem considered by Graham in his seminal paper [83] where he showed that list scheduling produces a $(2 - \frac{1}{m})$-approximate solution. In our speed scaling version, we make a standard assumption that there is a continuous function $P(s)$, such that if a processor is run at speed s, then its power, the amount of energy consumed per unit time, is $P(s) = s^{\alpha}$, for some $\alpha > 1$. For example, the well known cube-root rule for CMOS-based devices states that the speed s is roughly proportional to the cube-root of the power P, or equivalently, $P(s) = s^3$ (the power is proportional to the speed cubed) [132, 24]. Our second objective is to minimize the total energy consumed. Energy is power integrated over time. Thus we consider a bicriteria problem, in that we want to optimize both makespan and total energy consumption. Bicriteria problems can be formalized in multiple ways depending on

how one values one objective in relationship to the other. We say that a schedule S is a b-energy c-approximate if the makespan for S is at most cM and the energy used is at most bE where M is the makespan of an optimal schedule which uses E units of energy. The most obvious approach is to bound one of the objective functions and optimize the other. In our setting, where the energy of the battery may reasonably be assumed to be fixed and known, it seems perhaps most natural to bound the energy used, and to optimize makespan.

Power management for tasks with precedence constraints has received some attention in computer systems literature, see for example [86, 125, 163, 127] and the references therein. These papers describe experimental results for various heuristics.

In the last few years, interest in power management has seeped over from the computer systems communities to the algorithmic community. For a survey of recent literature in the algorithmic community related to power management, see [93].

8.2 Summary of results

For simplicity, we state our results when we have a single objective of minimizing makespan, subject to a fixed energy constraint, although our results are a bit more general.

We begin by noting that several special cases of $Sm \mid prec, energy \mid C_{max}$ are relatively easy. If there is only one processor ($S1 \mid prec, energy \mid C_{max}$), then it is clear from the convexity of $P(s)$ that the optimal speed scaling policy is to run the processor at a constant speed; if there were times where the speeds were different, then by averaging the speeds one would not disturb the makespan, but the energy would be reduced. If there are no precedence constraints ($Sm \mid energy \mid C_{max}$), then the problem reduces to finding a partition of the jobs that minimizes the ℓ_α norm of the load. A PTAS for this problem is known [6]. One can also get an $O(1)$-approximate constant-speed schedule using Graham's list scheduling algorithm. So for these problems, speed scaling doesn't buy you more than an $O(1)$ factor in terms of energy savings. Note that the $O(1)$ notation mentioned above means that the multiplicative factor is a constant that is independent of the input parameters even when they are taken into consideration.

We now turn to $Sm \mid prec, energy \mid C_{max}$. We start by showing that there are instances where every schedule, in which all machines have the same fixed speed, has a makespan that is a factor of $\omega(1)$ more than the optimal makespan. The intuition is that if there are several jobs, on different processors, that are waiting for a particular job j, then j should be run with higher speed than if it were the case that no jobs were waiting on j. In contrast, we show that what should remain constant is the aggregate powers of the processors. That is, we show that in any locally optimal schedule, the sum of the powers at which the machines run is constant over time. If the cube-root rule holds (power equals speed cubed), this means the sum of cubes of the machines speeds should be constant over time. We call schedules with this property *constant power schedules*. We then show how to reduce our energy minimization problem to the problem of scheduling on machines of different speeds (without energy considerations). In the three field scheduling notation, this problem is denoted by $Q \mid prec \mid C_{max}$. Using the $O(\log m)$-approximate algorithms from [34, 39], we can then obtain a $O(\log^2 m)$-energy $O(\log m)$-approximate algorithm for makespan for our problem. We then show a trade-off be-

tween energy and makespan for our problem. That is, an $O(b)$-energy $O(c)$-approximate schedule for makespan can be converted into $O(c \cdot b^{1/\alpha})$-approximate schedule. Thus we can then get an $O(\log^{1+2/\alpha} m)$-approximate algorithm for makespan.

We believe that the most interesting insight from these investigations is the observation that one can restrict one's attention to constant power schedules. This fact will also hold for several related problems.

8.2.1 Related results

We will be brief here, and refer the reader to the recent survey [93] for more details. Theoretical investigations of speed scaling algorithms were initiated by Yao, Demers, and Shankar [160]. They considered the problem of minimizing energy usage when each task has to be finished on one machine by a predetermined deadline. Most of the results in the literature to date focus on deadline feasibility as the measure for the quality of the schedule. Yao, Demers, and Shankar [160] give an optimal offline greedy algorithm. The running time of this algorithm can be improved if the jobs form a tree structure [121]. Bansal, Kimbrel, and Pruhs [14] and Bansal and Pruhs [16] extend the results in [160] on online algorithms and introduce the problem of speed scaling to manage temperature. For jobs with a fixed priority, Yun and Kim [161] show that it is NP-hard to compute a minimum energy schedule. In this model, priorities of jobs are given as part of the input, and an available job with the highest priority should be run at any time. They also give an FPTAS for the problem. Kwon and Kim [113] give a polynomial-time algorithm for the case of a processor with discrete speeds. Chen, Kuo and Lu [36] give a PTAS for some special cases of this problem. Pruhs, Uthaisombut, and Woeginger [137] give some results on the flow time objective function.

8.3 Formal problem description

The setting for our problems consists of m variable-speed machines. If a machine is run at speed s, its power is $P(s) = s^\alpha$, $\alpha > 1$. The energy used by each machine is power integrated over time.

An instance consists of n jobs and an energy bound E. All jobs arrive at time 0. Each job i has an associated work (or size) w_i. If this job is run consistently at speed s, it finishes in w_i/s units of time. There are precedence constraints among the jobs. If $i \prec j$, then job j cannot start before job i completes.

Each job must be run non-preemptively on some machine. The machines can change speed continuously over time. Although it is easy to see by the convexity of $P(s)$ that it is best to run each job at a constant speed.

A *schedule* specifies, for each time and each machine, which job to run and at what speed. A schedule is *feasible at energy level* E if it completes all jobs and the total amount of energy used is at most E. Suppose S is a schedule for an input instance I. We define a number of concepts which depend on S. The completion time of job i is denoted C_i^S. The makespan of S, denoted $C_{\max}^S$, is the maximum completion time of any job. A schedule is *optimal for energy*

level E if it has the smallest makespan among all feasible schedules at energy level E. The goal of the problem is to find an optimal schedule for energy level E. We denote the problem as $Sm \mid prec, energy \mid C_{\max}$.

We use s_i^S to denote the speed of job i. The execution time of i is denoted by x_i^S. Note that $x_i^S = w_i/s_i^S$. The power of job i is denoted by p_i^S. Note that $p_i^S = (s_i^S)^\alpha$. We use E_i^S to denote the energy used by job i. Note that $E_i^S = p_i^S x_i^S$. The total energy used in schedule S is denoted E^S. Note that $E^S = \sum_{i=1}^n E_i^S$. We drop the superscript S if the schedule is clear from the context.

8.4 No precedence constraints

As a warm-up, we consider the scheduling of tasks without precedence constraints. In this case we know that each machine will run at a fixed speed, since otherwise the energy use could be decreased without affecting the makespan by averaging the speed.

We may assume that there are at least as many jobs as there are machines. (If $m > n$, we simply ignore the last $m - n$ machines.) We then know that each machine will finish at the same time, since otherwise some energy from a machine which finishes early could be transferred to machines which finish late, decreasing the makespan. Furthermore there will be no gaps in the schedule.

For any schedule, denote the makespan by M, and denote the load on machine j, which is the sum of the work of the jobs on machine j, by L_j. Since each machine runs at a fixed speed, in this section we denote by s_j the speed of *machine* j, by p_j its power, and by E_j its energy used. By our observations so far we have $s_j = L_j/M$.

The energy used by machine j is

$$E_j = p_j M = s_j^\alpha M = \frac{L_j^\alpha}{M^{\alpha-1}} .$$

We can sum this over all the machines and rewrite it as

$$M^{\alpha-1} = \frac{1}{E}\sum_j L_j^\alpha. \tag{8.1}$$

It turns out that minimizing the makespan is equivalent to minimizing the ℓ_α norm of the loads. For this we can use the PTAS for identical machines given in [6]. Denote the optimal loads by $\text{OPT}_1, \ldots, \text{OPT}_m$. Similarly to (8.1), we have

$$\text{OPT}^{\alpha-1} = \frac{1}{E}\sum_j \text{OPT}_j^\alpha, \tag{8.2}$$

where OPT is the optimal makespan. For any $\varepsilon > 0$, we can find loads $L_1, \ldots, L_m$ in polynomial time such that $\sum_j L_j^\alpha \le (1+\varepsilon)\sum_j \text{OPT}_j^\alpha$. For the corresponding makespan M it now follows from (8.1) and (8.2) that

$$M^{\alpha-1} = \frac{1}{E}\sum_j L_j^\alpha \le (1+\varepsilon)\cdot\frac{1}{E}\sum_j \text{OPT}_j^\alpha = (1+\varepsilon)\text{OPT}^{\alpha-1}$$

or

$$M \leq (1+\varepsilon)^{1/(\alpha-1)}\text{OPT}.$$

Thus this gives us a PTAS for the problem $Sm \mid energy \mid C_{\max}$.

8.5 Main results

8.5.1 One speed for all machines

In the remainder of the paper, we only consider the case with precedence constraints. Suppose all machines run at a fixed speed s. We show that under this constraint, it is not possible to get a good approximation of the optimal makespan. For simplicity, we only consider the special case $\alpha = 3$.

Consider the following input: one job of size $m^{1/3}$ and m jobs of size 1, which can only start after the first job has finished. Suppose the total energy available is $E = 2m$. It is possible to run the large job at a speed of $s_1 = m^{1/3}$ and all others at a speed of 1. The makespan of this schedule is 2, and the total amount of energy required is $s_1^3 + m = 2m$.

Now consider an approximation algorithm with a fixed speed s. The total time for which this speed is required is the total size of all the jobs divided by s. Thus s must satisfy $s^3(m^{1/3} + m)/s \leq E = 2m$, or $s^2 \leq 2m/(m^{1/3} + m)$. This clearly implies $s \leq 2$, but then the makespan is at least $m^{1/3}/2$. Thus the approximation ratio is $\Omega(m^{1/3}) = \omega(1)$.

8.5.2 The power equality

To discuss the relationship among the powers of jobs in an optimal schedule, we need the following definitions. Given a schedule S of an input instance I, we define the *schedule-based constraint* $\prec_S$ among jobs in I as follows. For any jobs i and j, $i \prec_S j$ if and only if $i \prec j$ in I, or i runs before j on the same machine in S. Suppose S is a schedule where each job is run at a constant speed. The *power relation graph* of a schedule S of an instance I is a vertex-weighted directed graph $G = (V, E)$ created as follows:

- For each job i, create vertices u_i and v_i, each with weight p_i where p_i is the power at which job i is run. Vertex u_i corresponds to the *start* of job i. Vertex v_i corresponds to the *completion* of job i.
- In S, if $i \prec_S j$ and job j starts as soon as job i finishes (maybe on different machines), then create a directed edge (v_i, u_j).
- Two dummy vertices v_0 and u_{n+1} are added. In S, if job i starts at time 0, then create a directed edge (v_0, u_i). In S, if job i completes at time $C^S_{\max}$, then create a directed edge (v_i, u_{n+1}). Let $p_0 = \sum_{i:(v_0,u_i)\in E} p_i$, and let the weight of v_0 be p_0. Let $p_{n+1} = \sum_{i:(v_i,u_{n+1})\in E} p_i$, and let the weight of u_{n+1} be p_{n+1}.

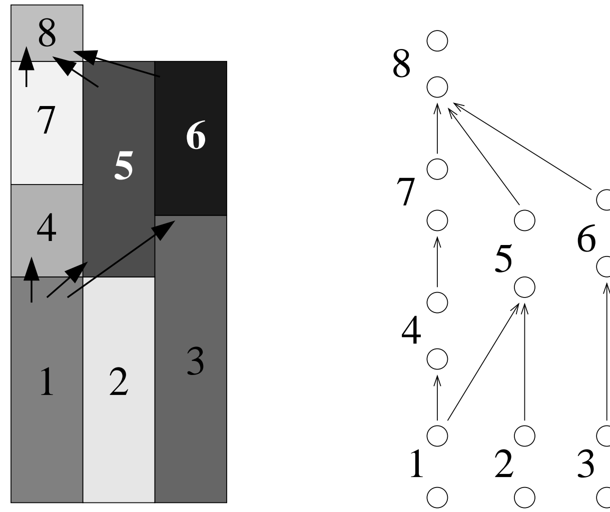


Figure 8.1: An example of a schedule and the corresponding power relation graph. In the schedule on the left, the arrows denote precedence constraints between jobs. Note that the precedence constraint between jobs 1 and 6 is not represented in the power relation graph. However, in the power relation graph there is an edge between jobs 2 and 5 since they run back to back on the same machine. In this example, the graph has six connected components.

Basically, the power relation graph G tells us which pairs of jobs on the same machine run back to back, and which pairs of jobs with precedence constraint $\prec$ between them run back to back. For an example, see Figure 8.1.

In this paper, we define a connected component of a directed graph G to be a subgraph of G that corresponds to a connected component of the underlying *undirected* graph of G. Note that an isolated vertex will form a connected component by itself. Suppose C is a connected component of a power relation graph G. Define $H(C) = \{u \mid (v, u) \in C\}$ and $T(C) = \{v \mid (v, u) \in C\}$. Note that $H(C)$ and $T(C)$ is the set of vertices at the heads and tails, respectively, of directed edges in C. If C contains only one vertex, then $H(C) = T(C) = \emptyset$. The completion of jobs in $T(C)$ and the start of jobs in $H(C)$ all occur at the same time. This holds simply because for each edge (v_i, u_j) in C, the completion time of job i is the starting time of job j by definition of an edge. Travelling through all the edges of a component shows that all completions and starts occur at a common time. If time t is when this occurs, we say that C *occurs at time* t. We say that a connected component C satisfies the *power equality* if

$$\sum_{i:u_i \in H(C)} p_i \quad = \quad \sum_{i:v_i \in T(C)} p_i$$

Note that p_i is the power at which job i is run, and is also the weight of vertices u_i and v_i. We say that a power relation graph G satisfies the *power equality* if each connected component of G has at least one edge, and each connected component of G satisfies the power equality. We now need to establish some properties of optimal schedules. The following observation is an obvious consequence of the convexity of the speed to power function.

Observation 8.1 *If S is an optimal schedule for some energy level E, then each job is run at a constant speed. This also implies that each job is run at a constant power.*

Lemma 8.1 *If S is an optimal schedule for some energy level E, then in the power relation graph G of S, each component contains at least one edge.*

Proof Let C be any connected component of the power relation graph G of an optimal schedule S. Assume to reach a contradiction that C contains no edges, that is, C contains only one vertex x. Let t be the time in S corresponding to the occurrence of C. Vertex x either corresponds to the start of some job i ($x = u_i$), or the completion of some job i ($x = v_i$).

If $x = u_i$ for some job i, then x corresponds to the start of some job i. Since there is no edge incident to u_i, then no jobs complete at time t. Thus, the machine that job i runs on is idle right before time t. We can modify the schedule S by starting job i earlier and running job i at a slower speed without violating the precedence constraints. Slowing down the job reduces the energy used. The energy saved could be reinvested elsewhere to get a better makespan. This contradicts the fact that S is optimal.

If $x = v_i$ for some job i, then x corresponds to the completion of job i. Since there is no edge incident from v_i, then no jobs start at time t. Thus, the machine that job i runs on is idle right after time t. We can modify the schedule S by running job i at a slower speed so that it completes later without violating the precedence constraints. Also this does not increase the makespan because job i could not be the last job to finish from the construction of G. Slowing down the job reduces the energy used. The energy saved could be reinvested elsewhere to get a better makespan. This contradicts the fact that S is optimal. □

Lemma 8.2 *If S is an optimal schedule for some energy level E, then the power relation graph G of S satisfies the power equality.*

Proof Let G be the power relation graph of an optimal schedule S. From Lemma 8.1, every component of G contains at least one edge. Thus, it only remains to show that each component of G satisfies the power equality. The idea of the proof is to consider an arbitrary component C of G. Then create a new schedule S' from S by slightly stretching and compressing jobs in C. Since S is optimal, S' cannot use a smaller amount of energy. By creating an equality to represent this relationship and solving it, we have that C must satisfy the power equality.

Now we give the details. C contains at least two vertices by Lemma 8.1. Let $\varepsilon \neq 0$ be a small number such that $x_i + \varepsilon > 0$ for any job i in $T(C)$, and $x_i - \varepsilon > 0$ for any job i in $H(C)$. Note that we allow ε to be either positive or negative.

We create a new schedule S' by modifying schedule S in the following manner. Increase the execution time of every job in $T(C)$ by ε, and decrease the execution time of every job in $H(C)$ by ε. All other jobs are unchanged. Note the following:
(1) The execution time of job i in $T(C)$ in S' is positive because $x_i + \varepsilon > 0$.
(2) The execution time of job i in $H(C)$ in S' is positive because $x_i - \varepsilon > 0$.
(3) For $|\varepsilon|$ small enough, S' has the same power relation graph as S.
In particular, we choose ε such that $|\varepsilon|$ is less than the smallest difference between two successive times at which connected components occur (i.e., at which the set of jobs being executed

changes). Therefore, S' is a feasible schedule having the same power relation graph as S. Observe that the makespan of S' remains the same as that of S. All that has changed is the timing of some inner changeover point.

As an example, in Figure 8.1 we might take the connected component consisting of v_1, v_2, u_4, u_5. Changing the execution times in this component as described above means that the horizontal line between jobs 1 and 2 and jobs 4 and 5 gets moved slightly up or down, without affecting the rest of the schedule. Our restriction (3) on ε means that this line is not for instance moved above the starting point of job 6, which would violate a precedence constraint and give an infeasible schedule.

The change in the energy used, $\Delta E(\varepsilon)$, is

$$\begin{aligned}\Delta E(\varepsilon) &= E^{S'} - E^{S} \\ &= \sum_{i:v_i \in T(C)} \left(E_i^{S'} - E_i^{S}\right) + \sum_{i:u_i \in H(C)} \left(E_i^{S'} - E_i^{S}\right) \\ &= \sum_{i:v_i \in T(C)} \left(\frac{w_i^\alpha}{(x_i+\varepsilon)^{\alpha-1}} - \frac{w_i^\alpha}{x_i^{\alpha-1}}\right) + \sum_{i:u_i \in H(C)} \left(\frac{w_i^\alpha}{(x_i-\varepsilon)^{\alpha-1}} - \frac{w_i^\alpha}{x_i^{\alpha-1}}\right)\end{aligned}$$

Since S is optimal, $\Delta E(\varepsilon)$ must be non-negative. Otherwise, we could reinvest the energy saved by this change to obtain a schedule with a better makespan. Since the derivative $\Delta E'(\varepsilon)$ is continuous for $|\varepsilon|$ small enough, we must have $\Delta E'(0) = 0$. We have

$$\Delta E'(\varepsilon) = \sum_{i:v_i \in T(C)} \frac{(1-\alpha)w_i^\alpha}{(x_i+\varepsilon)^\alpha} + \sum_{i:u_i \in H(C)} \frac{(\alpha-1)w_i^\alpha}{(x_i-\varepsilon)^\alpha}$$

Substitute $\varepsilon = 0$ and solve for $\Delta E'(0) = 0$.

$$\begin{aligned}\Delta E'(0) &= 0 \\ \sum_{i:v_i \in T(C)} \frac{(1-\alpha)w_i^\alpha}{x_i^\alpha} - \sum_{i:u_i \in H(C)} \frac{(1-\alpha)w_i^\alpha}{x_i^\alpha} &= 0 \\ \sum_{i:v_i \in T(C)} \frac{(1-\alpha)w_i^\alpha}{x_i^\alpha} &= \sum_{i:u_i \in H(C)} \frac{(1-\alpha)w_i^\alpha}{x_i^\alpha} \\ \sum_{i:v_i \in T(C)} s_i^\alpha &= \sum_{i:u_i \in H(C)} s_i^\alpha \\ \sum_{i:v_i \in T(C)} p_i &= \sum_{i:u_i \in H(C)} p_i\end{aligned}$$

Thus, this connected component C satisfies the power equality. Since C is an arbitrarily chosen connected component in G, then G satisfies the power equality, and the result follows. □

Note that the above proof also establishes that the power equality must also hold for any schedule that locally optimal schedule with respect to the change considered in the proof.

Let $p_i(t)$ be the power at which job j runs at time t. Let $p(k,t)$ be the power at which machine k runs at time t. By convention if job i starts at time t_1 and completes at time t_2, we say that it runs in the close-open interval $[t_1, t_2)$. If a job has just finished at time t and another has just start at time t on machine k, then $p(k,t)$ is equal to the power of the *starting* job. We will use $p(k,t^-)$ to denote the power of the *completing* job. Also by convention, if no job is running at time t on machine k, then $p(k,t) = 0$.

Lemma 8.3 *If S is an optimal schedule for some energy level E, there exists a constant p such that at any time t, $\sum_{k=1}^{m} p(k,t) = p$, i.e. the sum of the powers of all machines at time t is p.*

Proof Suppose S is an optimal schedule. Let $t_0 = 0$. For $i \geq 1$, let t_i be the earliest time, if it exists, strictly after t_{i-1} at which some job completes or starts. Suppose t_l is the completion time of the last job. For $i = 0, ..., l-1$ and for any time t' such that $t_i < t' < t_{i+1}$, we will show that

$$\sum_{k=1}^{m} p(k,t_i) = \sum_{k=1}^{m} p(k,t') \quad \text{and} \tag{8.3}$$

$$\sum_{k=1}^{m} p(k,t_i) = \sum_{k=1}^{m} p(k,t_{i+1}) \tag{8.4}$$

If this is the case, then the result follows.

Let i be an index such that $0 \leq i \leq l-1$. Let t' be any time such that $t_i < t' < t_{i+1}$. We now prove (8.3). Since no jobs start or complete in the interval $(t_i, t']$, then the same set of jobs are running at time t and t'. By Observation 8.1, each job runs at a constant speed at all time. This also means that each job runs at a constant power at all time. Thus, (8.3) follows.

We now prove (8.4). Let A be the set of jobs that are running (or have just started) at time t_i. Let B be the set of jobs that are running (or have just started) at time t_{i+1}. Since no jobs start or finish during (t_i, t_{i+1}), then $A - B$ is the set of jobs that completes at time t_{i+1}, $B - A$ is the set of jobs that starts at time t_{i+1}, and $A \cap B$ is the set of jobs that has been running since time t_i (or earlier) and until after t_{i+1}. If X is a set of jobs, then let $M(X)$ be the set of machines on which jobs in X run.

$$\begin{aligned}
\sum_{k=1}^{m} p(k,t_i) &= \sum_{j \in A} p_j(t_i) \\
&= \sum_{j \in A-B} p_j(t_i) + \sum_{j \in A \cap B} p_j(t_i) \\
&= \sum_{j \in A-B} p_j(t_{i+1}^-) + \sum_{j \in A \cap B} p_j(t_{i+1}) \quad \text{by the same argument as (8.3)} \\
&= \sum_{k \in M(A-B)} p(k, t_{i+1}^-) + \sum_{j \in A \cap B} p_j(t_{i+1}) \\
&= \sum_{k \in M(B-A)} p(k, t_{i+1}) + \sum_{j \in A \cap B} p_j(t_{i+1}) \quad \text{from Lemma 8.2}
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{j \in B-A} p_j(t_{i+1}) + \sum_{j \in A \cap B} p_j(t_{i+1}) \\
&= \sum_{j \in B} p_j(t_{i+1}) = \sum_{k=1}^{m} p(k, t_{i+1})
\end{aligned}
$$

□

8.5.3 Algorithm

Lemma 8.3 implies that the total power at which all the machines run is constant over time (only the distribution of the power over the machines may vary). We will describe a scheme to use this lemma to relate $Sm \mid prec, energy \mid C_{\max}$ to the problem $Q \mid prec \mid C_{\max}$. Then, we can use an approximation algorithm for the latter problem given in [34] to obtain an approximate schedule. The schedule is then scaled so that the total amount of energy used is within the energy bound E.

Let $\bar{p}$ be the sum of powers at which the machines run in the optimal schedule $\text{OPT}(I, E)$. Since energy is power times makespan, we have $\bar{p} = E/\text{OPT}(I, E)$. However, an approximation algorithm does not know the value of $\text{OPT}(I, E)$, so it cannot immediately compute $\bar{p}$. Nevertheless, we will assume that we know the value of $\bar{p}$. The value of $\bar{p}$ can be approximated using binary search, and this will be discussed later. Given $\bar{p}$, define the set $M(\bar{p})$ to consist of the following *fixed speed* machines: 1 machine running at power $\bar{p}$, 2 machines running at power $\bar{p}/2$, and in general 2^i machines running at power $\bar{p}/2^i$ for $i = 0, 1, ..., \lfloor \log(m+1) \rfloor - 1$. Denoting the total number of machines so far by m', there are an additional $m - m'$ machines running at power $\bar{p}/2^{\lfloor \log(m+1) \rfloor}$. Thus there are m machines in the set $M(\bar{p})$, but the total power is at most $(\log m + 1)\bar{p}$. We show in the following lemma that if the optimal algorithm is given the choice between m variable speed machines with total energy E and the set $M(\bar{p})$ of machines just described, where it is allowed to use preemptions, it will always take the latter, since the makespan will be smaller.

Lemma 8.4 *We have*

$$\text{PRMOPT}_{M(\bar{p})}(I) \leq \text{OPT}(I, E),$$

where $\text{PRMOPT}_{M(\bar{p})}(I)$ is the makespan of the optimal preemptive schedule using fixed speed machines in the set $M(\bar{p})$, and $\text{OPT}(I, E)$ is the makespan of the optimal schedule using m variable-speed machines with energy bound E.

Proof In an abuse of notation, we let $\text{PRMOPT}_{M(\bar{p})}(I)$ and $\text{OPT}(I, E)$ refer to the makespans of the two optimal schedules as well as those respective schedules themselves.

We will create a preemptive schedule S using fixed speed machines in the set $M(\bar{p})$. We will consider each time t and assign jobs in $\text{OPT}(I, E)$ to machines in S. We will show that the assignment can be feasibly done. We abuse the notation by using S to refer to the makespan of schedule S. Thus,

$$\text{PRMOPT}_{M(\bar{p})}(I) \leq S \leq \text{OPT}(I, E).$$

$FindSchedule(I, p)$

1. Find a schedule for instance I and machines in the set $M(p)$ using the algorithm from Chekuri and Bender [34].
2. Reduce the speed of all machines by a factor of $\log^{2/\alpha} m$
3. Return the resulting schedule.

$\text{ALG}(I, E)$

1. Set $p^* = \left(\frac{E}{W}\right)^{\frac{\alpha}{\alpha-1}}$ where W is the total work of all jobs.
2. Using binary search on $[0, p^*]$ with p as the search variable, find the largest value for p such that this 2-step process returns true. Binary search terminates when the binary search interval is shorter than 1.
   (a) Call $FindSchedule(I, p)$.
   (b) If for the schedule obtained we have $\sum_{i=1}^{n} s_i^{\alpha-1} w_i \leq E$, return true

Figure 8.2: Our speed scaling algorithm. The input consists a set of jobs I and an energy bound E.

Consider any time t in $\text{OPT}(I, E)$. Denote the power of machine k of $\text{OPT}(I, E)$ at this time by P_k. Suppose the machines are labeled so that $P_1 \geq P_2 \geq ... \geq P_m$. Now we simply assign the job on machine 1 to the machine of power $\bar{p}$ in S. And for $i \geq 1$ we assign the jobs on machines $2^i, \ldots, 2^{i+1} - 1$ to the machines of power $\bar{p}/2^i$ in S.

Clearly, $P_1 \leq \bar{p}$, since no machine can use more than $\bar{p}$ power at any time. In general, we have that

$$P_j \leq \bar{p}/j \text{ for } j = 1, \ldots, m.$$

If we can show that the first machine in any power group has at least as much power as the corresponding machine of $\text{OPT}(I, E)$, this holds for all the machines. But since machine 2^i in S has power exactly $\bar{p}/2^i$, this follows immediately.

It follows that S allocates each individual job at least as much power as $\text{OPT}(I, E)$ at time t. We can apply this transformation for any time t, where we only need to take into account that S might finish some jobs earlier than $\text{OPT}(I, E)$. So the schedule for S might contain unnecessary gaps, but it is a valid schedule, at least when we allow preemptions. This proves the lemma. □

To construct an approximate schedule, we assume the value of $\bar{p}$ is known, and the set of fixed speed machines in $M(\bar{p})$ will be used. The schedule is created using the algorithm given in [34]. The schedule created may use too much energy. To fix this, the speeds of all jobs are decreased so that the total energy used is within E at the expense of having a longer makespan. The steps are given in subroutine $FindSchedule$ in Figure 8.2.

8.5.4 Analysis

Lemma 8.5 *Suppose* $p = E/OPT(I,E)$. *Subroutine* $FindSchedule(I,p)$ *creates a schedule which has makespan* $O(\log^{1+2/\alpha} m)\text{OPT}(I,E)$ *and uses energy* $O(E)$.

Proof Let S_1 and S_2 denote the schedules obtained in steps 1 and 2 of $FindSchedule(I,p)$, respectively. Schedule S_2 is the one returned by $FindSchedule$. First we analyze the makespan.

From the results in [34],

$$C_{\max}^{S_1} = O(\log m)\text{PRMOPT}_{M(p)}(I).$$

This holds because although their algorithm does not use preemptions, it has this approximation ratio even when compared against an optimal preemptive algorithm. In step 2, the speed of every job decreases by a factor of $\log^{2/\alpha} m$. Thus, the makespan increases by a factor of $\log^{2/\alpha} m$. From Lemma 8.4, $\text{PRMOPT}_{M(p)}(I) \leq \text{OPT}(I,E)$. Therefore, taken together, we have

$$\begin{aligned} C_{\max}^{S_2} &= (\log^{2/\alpha} m)C_{\max}^{S_1} \\ &= (\log^{2/\alpha} m)O(\log m)\text{PRMOPT}_{M(p)}(I) \\ &= O(\log^{1+2/\alpha} m)\text{OPT}(I,E). \end{aligned}$$

Next we analyze the energy. The machines in the schedule $\text{OPT}(I,E)$ run for $\text{OPT}(I,E)$ time units at the total power of $p = E/\text{OPT}(I,E)$ consuming a total energy of E. Recall that if all machines in $M(p)$ are busy, the total power is at most $p(1+\log m)$.

Schedule S_1 runs the machines for $O(\log m)\text{PRMOPT}_{M(p)}(I)$ time units at the total power at most $p(1+\log m)$. Thus, it uses energy at most

$$\begin{aligned} & p(1+\log m)\,O(\log m)\text{PRMOPT}_{M(p)}(I) \\ \leq\ & O(\log^2 m)\,p\,\text{OPT}(I,E) = O(\log^2 m)\,E \end{aligned} \tag{8.5}$$

where the inequality follows from Lemma 8.4. The speeds at which the machines in S_2 run are $\log^{2/\alpha} m$ slower than those in $M(p)$, which S_1 uses. Thus, the total power at which the machines in S_2 run is $\log^2 m$ times smaller than that of S_1. By (8.5), this is $O(E)$. □

Note that when we decrease the speed in S_2 by some constant factor, the makespan increases by that factor and the energy decreases by a larger constant factor. To find the value of $\bar{p}$, we use binary search in the interval $[0, p^*]$ where p^* is an initial upper bound to be computed shortly. We continue until the length of the interval is at most 1. We then use the left endpoint of this interval as our power. Now we compute the initial upper bound p^*. For a given schedule, the total energy used is

$$\begin{aligned} \sum_{i=1}^{n} p_i x_i &= \sum_{i=1}^{n} s_i^{\alpha} w_i / s_i \\ &= \sum_{i=1}^{n} s_i^{\alpha-1} w_i. \end{aligned}$$

The best scenario that could happen for the optimal algorithm is when the work is evenly distributed on all the machines and all the machines run at the same speed at all time. Let W be the total work of all the jobs. Completing x units of work at a speed of s requires $s^{\alpha-1}x$ units of energy. If each of the m machines processes W/m units of work, then it takes a total $Ws^{\alpha-1}$ units of energy. This must be less than E. For the speed we find $s^{\alpha-1} \leq E/W$ and thus $p^{\frac{\alpha-1}{\alpha}} \leq E/W$. This gives us an initial upper bound for p for the binary search:

$$p \leq p^* = \left(\frac{E}{W}\right)^{\frac{\alpha}{\alpha-1}}.$$

OPT does not use a higher power than this, because then it would run out of energy before all jobs complete.

From Lemma 8.5 and our analysis above, the following theorem holds.

Theorem 8.1 ALG *is an* $O(\log^{1+2/\alpha} m)$*-approximation algorithm for the problem* $Sm \mid prec$ $energy \mid C_{\max}$ *where the power is equal to the speed raised to the power of* α *and* $\alpha > 1$.

Chapter 9

Real-time integrated prefetching and caching

In this chapter, we present a new theoretical model for real-time prefetching and caching. Interestingly, there is very little theoretical work on the real-time setting of this problem. We are only aware of [65] which covers parallel disk prefetching in a read-once setting without caching. This is astonishing, since real-time properties are essential for more and more important applications such as games, virtual reality, graphics animations, or multimedia. Memory hierarchies get more and more important for these applications since larger and larger data set are considered and since mobile devices have only very limited fast memory.

In Section 9.1, we propose to model real time aspects by associating a time windows with each request during which it needs to be in cache. As before we do prefetching and caching, and as soon as a fetch for a page starts, this page occupies one slot in the cache. We believe that our model may be a more accurate representation of the situation in practice. In particular, time windows rather than just deadlines allow us to efficiently model the amount of time a data block is needed for processing and we can also require several blocks to be available concurrently. The only simpler model we could think of would use unit time windows. But then we would need many repeated requests to model longer time windows. This would lead to exponentially longer problem descriptions in the worst case.

We fist prove some generally useful properties of the problem in Section 9.2. In Section 9.3 we present our algorithm REALMISER which uses a "semi"-greedy approach (REALMISER). REALMISER uses the frequently used basic trick to build the schedule backward in time [108, 91]. Apart from this it is a new algorithm however. Its main invariant is that it uses as little space as possible at all times and in order to achieve that it has to move previously scheduled requests. The algorithm therefore has quadratic worst case performance. Moreover it is I/O-optimal, i.e., it does not perform more fetches than necessary.

We additionally consider an online version of our problem in Section 9.4. In our online model, algorithms have a certain amount of lookahead, input arrives incrementally, and a partial solution needs to be determined without knowledge of the remaining input. We say that an algorithm has a *lookahead* of ℓ if at time t, it can see all requests for pages that are required no later than at time $t + \ell$. In the *resource augmentation* model, the online algorithm has more

resources than the offline algorithm that it is compared to. There are several ways to give an online algorithm more resources in the current problem: it can receive a larger cache, a faster disk (so that fetches are performed faster) or a combination of these.

We show that competitive algorithms are possible using resource augmentation on the speed *and* lookahead, and we provide a tight relationship between the amount of resource augmentation on the speed and the amount of lookahead required.

Section 9.5 concludes with a short summary and some possible future questions.

More Related Work In the model introduced by Cao et al. [27] and further studied by Kimbrel and Karlin [108] and Albers et al. [4], the requests are given as a simple sequence without an explicit notion of time. It is assumed that serving a request to a page residing in cache takes one time unit, and fetching a page from disk takes F time units. When a fetch starts and the cache is full, a page must be evicted. If a page is not in cache when it is required, the processor must wait (stall) until the page has been completely fetched. The goal is to minimize the processor stall time.

Thus in this model, pages have implicit deadlines in the sense that each page should be in cache exactly one time unit after the previous request. However, when the processor incurs stall time, these implicit deadlines are shifted by the amount of stall time incurred. Additionally, this model does not cover the cases where many pages are required in a small time interval and conversely, where more time may elapse between two successive requests. Nor is it possible to model the case where a page is required over a certain time interval.

Albers [2] considers the impact of lookahead in the classical non-real-time situation. She shows that in order to be useful in a worst case sense, lookahead has to be measured in terms of the number of distinct pages referred to in the lookahead. This can be a problem in practice since very long lookahead sequences might be required if some blocks are accessed again and again. In our real-time setting the situation is different and very natural — we can measure lookahead in terms of time.

9.1 Problem definition

We consider the problem of prefetching pages into a cache of fixed size k. The request sequence σ serves as input and consists of pairs $\sigma_i := (p_i, [d_i, e_i))$ denoting a page and the interval in which it must reside in the cache. The d_i are also denoted *deadlines*, the e_i are the *ends of intervals*. Without loss of generality we may assume the input is sorted such that $d_1 \leq \ldots \leq d_n$ for $n = |\sigma|$. Transferring a page to the cache takes time 1. The earliest possible fetch time is $t = 0$.

The output is given by a sequence $f_1, \ldots f_n$ of fetch times for the corresponding requests. One cache slot is occupied by p_i in the time interval $[f_i, e_i)$, and possibly longer. Multiple requests of the same page can be served by the same fetch, the page must then reside in the cache until the last of these requests is served. A feasible schedule must satisfy $\forall i \in \{1, \ldots, n\}$: $f_i + 1 \leq d_i$ to match the real-time requirements. In addition to this, the cache must be sufficient,

i. e. $\forall t : |\{f_i : f_i \le t < e_i\}| \le k$, and the disk must not be overcommitted — $\forall t : |\{p_i : t \in [f_i, f_i + 1)\}| \le 1$.

9.2 Problem properties

Definition 9.1 ***(FIFO property)*** *A schedule satisfies the* FIFO Property *if, when it has a fetch at time f for a page that is next required at time d, there is no later fetch which loads a page that is required before time d. More formally, this is the case if and only if $\forall i, j \in \{1, \ldots, n\} : (f_i \le f_j) \Rightarrow (\exists i' : i' \le j \wedge f_{i'} = f_i)$.*

If algorithm ALG constructs a schedule that fulfills the FIFO property, we write $FIFO(\text{ALG})$.

Definition 9.2 ***(BUSY property)*** *A schedule satisfies the* BUSY Property *if, when it has a fetch at time f for a page that is next required at time d, it defines fetches at all times $f + i$ for $i = 1, 2, \ldots, \lceil d - f \rceil - 1$.*

Lemma 9.1 *There exists an I/O-optimal schedule with both the FIFO property and the BUSY property.*

Proof Consider an arbitrary I/O-optimal schedule. We can make two local improvements:

1. If page p is fetched before page q, but after the fetch for p the first deadline for q occurs before the first deadline to p, we can switch these two fetches without violating any deadlines or the cache capacity (the amount of slots occupied by p and q remains the same, we only load a different page first).

2. If a fetch for page p ends at time t_1, but page p is first requested at time $t_2 > t_1$, and moreover no new fetch starts until time $t_3 > t_1$, we can move this fetch for p forward until it ends at time $\min(t_2, t_3)$.

Since these improvements do not increase the number of fetches but only move them, the schedule always remains I/O-optimal. It is now easy to show using backward induction that if we can not make any local improvement to an optimal schedule, it satisfies the FIFO property and the BUSY property. □

9.3 Algorithm REALMISER

In this section we present the algoritm REALMISER for the real-time integrated prefetching and caching problem. This algorithm works by working backwards from the last deadline. For each new request, it modifies the existing schedule to maintain I/O-optimality. While the schedule is being constructed, our algorithm keeps track of a value NOW which is the time at which the earliest fetch of the current schedule starts. REALMISER uses a predicate which is called INFEASIBLE and is defined as follows.

INFEASIBLE= (NOW < 0 or there exists a time at which there are at least $k+1$ pages in cache).

The algorithm itself is defined in Figure 9.1. It uses the following definition. We also define a concept called *slack* which will be important later.

Definition 9.3 *A* tight fetch *is a fetch which starts one time unit before the page is requested, i. e., at the latest possible time.*

Definition 9.4 *The* slack *of a fetch is the amount of time between the end of the fetch and the corresponding deadline.*

Set NOW $:= d_n - 1$. For $i = n, \ldots, 1$, do the following.

1. Define a fetch for the ith request, say page p_i with deadline d_i, at time NOW. Let NEXT be the next time that the page p_i is fetched at or after time NOW. If there is no such time, then if INFEASIBLE, output FAIL, else, set NOW := NOW -1, evict page p_i at time e_i and stop (i. e. stop processing request i).
2. If NEXT $\leq d_i - 1$, remove the fetch that was just defined at time NOW and stop.
3. If NEXT $> e_i$, consider the interval $[e_i, \text{NEXT}]$. Let FULL be the first time the cache is full in this interval. If this does not happen, set FULL $= \infty$. If there is a tight fetch which finishes in the interval [FULL,NEXT], then
   (a) evict p_i at time e_i
   (b) if INFEASIBLE, output FAIL, else, set NOW := NOW -1 and stop.
4. We are in the case NEXT $\in (d_i - 1, e_i]$, or NEXT $> e_i$ but the condition in the previous step does not hold. Remove the fetch at time NEXT. For each fetch before time NEXT in order of decreasing starting time, move the fetch forward as much as possible without violating its deadline or overlapping with the next fetch after it. Set NOW equal to the earliest fetch time in the resulting schedule and subtract 1.

Figure 9.1: Algorithm REALMISER

In Step 2, we do not need to check feasibility because we do not change the schedule. We will prove in the following that we also do not need to check it in Step 4. Note that the condition in Step 2 can easily be satisfied even though deadlines are sorted: if a fetch for a page p_i is defined a long time before d_i, p_i might be requested again between this fetch and time d_i. We give an example of the execution of REALMISER for $k = 2$ in Figure 9.2.

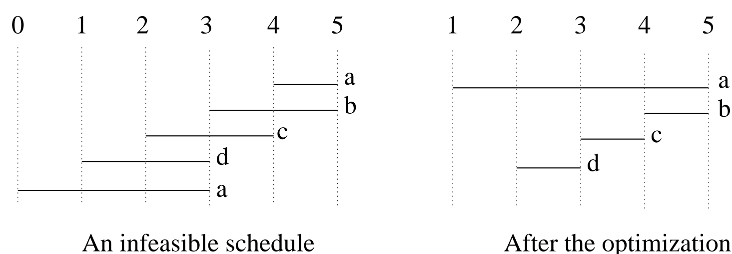


Figure 9.2: Suppose $k = 2$. REALMISER treats the requests in order of nondecreasing deadline. Horizontal lines represent a cache location which is occupied by the page at the end of this line (which is where this page is needed). When REALMISER gets to the request for page a at time 3, it initially creates the infeasible schedule on the left: there are three pages in cache in the interval $[2, 3]$. However, REALMISER then removes the fetch for page a at time 4. The other fetches move forward by 1 time unit, and the resulting schedule is feasible.

9.3.1 Analysis of REALMISER

We say that a cache is *full* if there are no empty slots in the cache (recall that a page occupies a slot as soon as it starts being fetched) and moreover all pages in the cache are still needed at some point in the future. Pages that are evicted at time t are not taken into consideration to determine whether the cache is full or not at time t. For convenience, we consider fetches to be half-open intervals of the form $(f, f + 1]$.

Lemma 9.2 *The schedule of* REALMISER *has the FIFO property and the BUSY property at all times during the execution of* REALMISER. *Specifically, for any fetch, this holds both before and after the optimization that RealMiser performs on this fetch.*

Proof By backwards induction, starting with the last request. For the last request, there is nothing to prove (if its deadline is at time d, its fetch is scheduled at time $d-1$, at least initially).

Consider an earlier request i, for page p needed at time d. The first time that RealMiser schedules a fetch for p, it only considers the current schedule, based on requests $i + 1, \dots, n$, which have the same or later deadlines. The initial definition of the new fetch is one time unit before the earliest current fetch. Say the earliest current fetch starts at time $f + 1$. This is for request $i + 1$, which is needed at time $d' \geq d$ because this is how the input is ordered by RealMiser. By induction, fetches start at all times $f + 1 + j$ for $j = 1, 2, \dots, \lceil d' - (f + 1)\rceil - 1$. We have $f + 1 + \lceil d' - (f + 1)\rceil - 1 = f + \lceil d' - f\rceil - 1 \geq f + \lceil d - f\rceil - 1$. Thus the FIFO property and the BUSY property hold before the optimization for request i.

If the fetch is immediately removed again in Step 2, we are done by induction. This is also true if REALMISER does not change the schedule (does not execute Step 4). In Step 4, the next fetch to p is removed and all or some earlier fetches are moved forward as far as possible, including the earliest one for p. Thus the BUSY property holds again. Moreover, RealMiser will never reorder existing fetches. Thus the FIFO property still holds as well. □

Lemma 9.3 *All fetches that RealMiser defines occur at times of the form $d_n - j$, where j is an integer, unless there exists a time interval after the earliest fetch and before d_n where no page is*

being fetched.

Proof This follows immediately from the BUSY property. If pages are being fetched at all times during the execution, RealMiser schedules these at times $d_n - j$, since it schedules each fetch as late as possible. If this is not the case, there is clearly some interval where no pages are being fetched. □

Theorem 9.1 *If there exists a feasible schedule, RealMiser computes an I/O-optimal one.*

Proof Let $\text{ALG}(i)$ be the number of fetches that algorithm ALG defines to serve the sequence consisting only of the requests $i, \ldots, n$. Denote an optimal algorithm by OPT, and without loss of generality assume it satisfied the FIFO and BUSY properties. We use a proof by induction.

Hypothesis: The schedule of RealMiser is I/O-optimal for any input consisting of at most i requests that allows a feasible schedule, and there is no feasible schedule that fetches its first page later.

Base case: Consider an input consisting of a single request. RealMiser defines a single fetch for it, at the last possible time. Thus, RealMiser is I/O-optimal, and no schedule can start its first fetch later than RealMiser.

Induction step: Consider the requests $i, \ldots, n$. By the induction hypothesis, RealMiser is optimal for an input that contains only the requests $i + 1, \ldots, n$. (We assume that a feasible schedule for the input exists, so it certainly exists for any subset of the input). Consider request i, for page p_i. We abbreviate p_i by p in this proof. We need to check the following properties:

T1. RealMiser does not define more than the optimal number of fetches

T2. The first fetch cannot start later in any schedule

T3. The schedule created by RealMiser is feasible.

Property T2 follows immediately, because in all cases RealMiser either puts the first fetch immediately before the next fetch (so we can use induction, since the optimal schedule also satisfies the FIFO property), or one time unit before its deadline (so it clearly cannot be postponed). This also means that if RealMiser defines a fetch before time 0, there does not exist a feasible schedule.

RealMiser starts by defining a fetch for p at time NOW, where NOW $+1$ is the earliest current fetch time. It then checks whether any optimization is possible. We now consider the execution of RealMiser step by step.

Step 1 If REALMISER stops in Step 1, then p is not requested again later. This implies immediately that T1 holds, because REALMISER defines one extra fetch and this is the best you can do. Also the new schedule is feasible if the old one was (using induction and T2) and the instance allows a feasible schedule. This proves T3.

Step 2 If NEXT $> d_i - 1$, the fetch is immediately removed again and we are done by induction: we have the previous schedule again, for which the desired properties hold.

Suppose page p is again fetched later. After time NOW, say that this happens for the first time at time NEXT (in the current schedule). Thus p is required at time NEXT $+1$ or later. The question is whether we can afford to keep p in the cache in the interval [NOW,NEXT].

Step 4, case 1 If NEXT $\leq e_i$, we have no choice: we simply must keep p in cache until the end of the interval that we fetch it for. So in this case the fetch at time NEXT can certainly be removed. RealMiser executes step 4 in this case. It removes the fetch at time NEXT, so T1 holds by induction. The new schedule also satisfies the FIFO and BUSY property. Thus if it is infeasible, no feasible schedule exists by induction.

Now assume that NEXT $> e_i$, so again Step 4 is executed. Removing the access to p at time NEXT means that all fetches in the interval [NOW $+1$, NEXT] can be postponed by one time unit, unless one of those fetches had slack less than 1.

Step 4, case 2 If all fetches in [NOW $+1$, NEXT] have **slack at least 1**, there can be no interval in [NOW $+1$, NEXT] in which no fetch takes place (then the fetch immediately before that interval could be postponed, and RealMiser would have done this). Thus at all times, some page is being fetched. In this case all fetches can be postponed by 1 (after removing the access at time NEXT) without violating any deadlines, meaning that we can save one cache slot in every time step, and we can save one access by keeping p in cache. An example of this situation can be seen in Figure 9.2. It is clear that the resulting schedule is I/O-optimal (T1) and feasible (T3).

Step 4, case 3 If the **cache is nowhere full** during [NOW $+1$, NEXT], we can keep p in cache and save an access. The schedule for the other pages may remain the same, or some fetches may now occur later. Similar, if the cache is only full **after** the last fetch with slack less than 1, all the fetches after that one can be tightened as above after removing the access at time NEXT. Clearly, the new schedule does not violate cache capacity constraints or deadlines. If NOW < 0, then no feasible schedule exists.

In all these cases, we find

$$\text{REALMISER}(i) = \text{REALMISER}(i+1) = \text{OPT}(i+1) = \text{OPT}(i)$$

(the last equality follows since $\text{OPT}(i+1) \leq \text{OPT}(i) \leq \text{REALMISER}(i)$). So T1 holds if RealMiser executes step 4.

Step 3 We also have the following lemma which shows that in the remaining cases, RealMiser can simply evict page p after the end of its inverval and still be optimal. So T1 also holds if it stops in Step 3.

Lemma 9.4 *Let a fetch of page p (request i) start at time* NOW. *Suppose that p is again fetched later, and that this happens for the first time at time* NEXT. *Suppose there is at least one tight*

*fetch in the interval [*NOW $+1$, NEXT*], and denote the last time at which a tight fetch finishes by* TIGHT*. If the cache is full at some point no later than* TIGHT*, then* OPT$(i) =$ OPT$(i+1)+1$.

Proof If the cache is full at time FULL $\in$ [NOW+1,TIGHT], there are two simple cases. The third, more difficult case follows below. We call the fetch which is running at time FULL the *current* fetch.

1. k different pages are requested within an interval of length strictly less than 1 starting at time FULL. By assumption, all these pages are different from p. This means that no algorithm can keep p in cache until its next fetch, so the request sequence that includes p forces an extra fetch, and OPT$(i) =$ OPT$(i+1)+1$.

2. All pages in the cache are either required during the current fetch, or before (these pages were kept in cache to save a fetch on them). Suppose RealMiser has k' pages loaded at time FULL that are needed only after the current fetch (i. e. $k-k'$ pages are needed during the current fetch). This means that RealMiser has saved k' accesses to those pages.

   Suppose OPT(i) keeps p in its cache throughout the interval [NOW+1,NEXT]. Then one of these k' pages must be evicted by it, and later loaded again. However, RealMiser has the optimal number of fetches for the sequence $i+1,\ldots,n$. We have that OPT(i) has one fetch more than RealMiser for the requests $i+1,\ldots,n$. Thus OPT$(i) =$ REALMISER$(i+1)+1 =$ OPT$(i+1)+1$.

The only tricky case is where the cache is full at time FULL $\in$ [NOW+1,TIGHT], but some pages are already loaded to satisfy future requests (and not because they were requested before). This means that there are three sets of pages at time FULL:

1. k_1 pages required during the current fetch
2. k_2 pages already requested before, still needed after the current fetch
3. k_3 pages that are needed only after time FULL.

Of course, $k_1+k_2+k_3=k$, above we treated the case $k_3=0$. Let FULL now be the *last* time the cache is full in (NOW+1,NEXT]. If $k_3>0$, some pages are loaded that are needed only later.

Suppose that in the optimal schedule, p is in the cache throughout [NOW+1,NEXT], so definitely at time FULL. Then at least one of the pages that RealMiser has in the cache at time FULL, say q, must be missing in the optimal cache, since the cache of RealMiser is full. Page q is not requested during the current fetch, but there is a later need for it. If RealMiser is saving an access on page q since it was requested before time FULL, we are done: the optimal schedule still needs to load q, and without the request for p there exists a schedule with one less fetch for q (namely, the one of RealMiser), so OPT$(i) =$ OPT$(i+1)+1$. Otherwise, q is loaded only to satisfy a future request. In this case, consider the time starting from FULL.

Consider the pages that are loaded purely to satisfy requests after the current fetch. Say that the last time such a page is first needed (after FULL) is time t_1, and denote that page by p_1. (Possibly $p_1=q$.) At time t_1, we can make a similar division into sets as above. If any pages are in cache at time t_1, but the interval for which they were loaded has already expired, we can find

a time $t_2 > t_1$ where the last such page (say p_2) is first needed. We can continue this process until some time $t_\ell =: t^*$, which is defined as the earliest possible time such that all pages in cache were fetched to satisfy a request at or before time t^*. (Some of these pages might have been kept in cache to also satisfy later requests.)

We claim that RealMiser fetches pages continuously in the interval $[\text{FULL}, t^*]$. This follows simply by applying the BUSY property (Lemma 9.2) at time FULL, $t_1, t_2, \ldots$ successively until time t^* is reached. Consider the last fetch that starts before time t^*. By definition of t^*, this fetch cannot be for a page that is requested only after t^*. Thus this fetch must in fact be tight (no slack). Therefore

$$t^* \leq \text{TIGHT} \leq \text{NEXT}.$$

Since the cache is not full in the interval $[\text{FULL}, t^*]$, RealMiser never fetches the same page twice during $[\text{FULL}, t^*]$. All of these pages are needed in $[\text{FULL}, t^*]$, including q. Denote the set of fetched pages in $[\text{FULL}, t^*]$ by S.

The optimal schedule must load q at some point. So it must load at least one page $p^{(1)}$ that RealMiser loads in the interval $[\text{FULL}, t^*]$ already before this interval, since there is no time to fetch $|S| + 1$ pages. This implies $p^{(1)}$ is not required during the fetch which runs at time FULL, and RealMiser does not have either p or $p^{(1)}$ in its cache at time FULL. Therefore RealMiser has yet another page $q^{(1)}$ in cache that the optimal schedule does not have, since its cache is full.

We can now repeat this reasoning: if RealMiser saves a request on $q^{(1)}$ since it was already requested before, we immediately have $\text{OPT}(i) = \text{OPT}(i+1) + 1$ (the optimal schedule must still pay for $q^{(1)}$). Otherwise, we again find that the optimal schedule must load $q^{(1)}$ in the interval $[\text{FULL}, t^*]$, leading to yet another page $p^{(2)}$ that it must load before the interval due to time constraints.

Each such page $p^{(i)}$ implies an additional distinct page $q^{(i)}$ that RealMiser has in its cache at time FULL which the optimal schedule does not have, because the pages $p, p^{(1)}, \ldots, p^{(i)}$ are all in the cache of the optimal schedule at time FULL and the cache of RealMiser is full. (Each time we find that RealMiser does not have $p^{(i)}$ in its cache, because this page is requested in the interval $[\text{FULL}, t^*]$ and we know that RealMiser loads it after time FULL: if $p^{(i)}$ were already in the cache at time FULL, RealMiser would never drop it since the cache is not full afterwards.)

Finally, after at most k steps we either run out of pages and find a contradiction, or we find a page that RealMiser saves a request on and that the optimal schedule must pay for, implying that

$$\text{OPT}(i) = \text{REALMISER}(i+1) + 1 = \text{OPT}(i+1) + 1.$$

□

This Lemma immediately implies that T1 holds if RealMiser stops in Step 3 for this request. It is moreover clear that the new schedule does not violate deadlines. Thus if the new schedule is infeasible, no feasible schedule can exist by induction: either the cache capacity constraint is violated in all possible schedules, but then the instance does not admit a feasible schedule because the schedule starting from the next request was I/O-optimal and busy, or $\text{NOW} < 0$, but the next fetch already started as late as possible by induction. □

9.4 Online algorithms

In the pure online model, it is impossible for an online algorithm to handle the hard deadlines properly. In fact, we have the following lemma which shows that even lookahead does not help much.

Lemma 9.5 *Any finite amount of lookahead is insufficient by itself to provide feasible schedules.*

Proof Let the lookahead be $n-1$ time units. Consider the following request sequence.

Page	a	b	x_1	x_2	...	x_n	a (or b)
Deadline	1	2	3	4	...	$n+2$	$n+2$

At time 2 the online algorithm needs to decide whether it removes page a or page b from its cache to fetch x_1. However, with a lookahead of $n-1$, it is impossible to know which page to evict. □

The explanation is that an online algorithm cannot handle more than 1 pages being requested per time unit on average, because it will need to decide which pages to evict and will inevitably make the wrong decisions. We therefore consider the resource augmentation model.

An option is to give the online algorithm a larger cache than the offline algorithm it is compared to. However, the above example also shows that a larger cache does not really help: at some point a page must be evicted, and this will be the page on which the algorithm fails later.

We can also allow the online algorithm to fetch pages faster than the offline algorithm. We show in the following that this does allow for a competitive algorithm.

In particular, we show that using a very simple algorithm, we can handle any sequence of requests which allows a feasible schedule as long as we have a lookahead of k and can fetch pages with twice the speed of the offline algorithm. Equivalently, we can also give the online algorithm the power to fetch two pages at the same time, by assuming that it has two parallel disks that both store all the data that is required.

The next assumptions on how the optimal offline algorithm behaves simplify the analysis.

Global assumption 4 *No pages are evicted during a fetch.*

It can be seen that evicting pages during fetches, instead of waiting until the end of the current fetch and then evicting them, cannot help an algorithm with respect to deadlines of later requests.

Global assumption 5 *At most one page is evicted at the start of a fetch.*

Since at most one page can be loaded during a single fetch, it does not help to evict more than one page at the start of a fetch, since you can only fill one slot with this fetch anyway. On the other hand, it also does not harm to keep as many pages as possible in the cache, since you do not need more than one free slot for a fetch.

Global assumption 6 *Pages are evicted only at the start of fetches.*

Since at the beginning the cache is empty, and each fetch loads only one page, there is no need to evict pages at any other time.

Lemma 9.6 *The contents of the cache of the optimal offline algorithm change for at most one slot in any (half-open) interval of length 1.*

Proof This follows immediately from the above assumptions. □

Lemma 9.7 *In an instance that allows a feasible solution, in an interval I of length strictly smaller than $i \in \mathbb{N}$, there cannot be more than $k + i - 1$ requests for distinct pages.*

Proof At the deadline d of the last request in I, before any page is evicted, by the above assumptions k of the requested pages in I are in the offline cache. During any fetch, the configuration of the offline cache does not change. By Lemma 9.6 and Assumption 3, the configuration only changes at the end of fetches, and only by one page. Thus in I, the configuration can only change at most $i - 1$ times before the final fetch. This means that in total, at most $k + i - 1$ distinct pages are present in the cache at some point during I. This is then an upper bound for the number of pages that can be requested in a feasible problem instance. □

Remark 9.1 *It is also clear that in any interval $(0, t]$, at most $\lfloor t \rfloor$ distinct pages can be requested in a feasible instance.*

Consider the greedy algorithm for general k. This algorithm simply loads pages in the order in which they are requested, as early as possible, and evicts pages that it does not see in its lookahead.

Lemma 9.8 *Greedy with a speed of s and a lookahead of $k/(s-1)$ creates a feasible schedule for each input for which a feasible schedule exists.*

Proof We prove by induction that the greedy algorithm provides a feasible schedule, if one exists. Suppose the algorithm sees its first request at time t. Then this request has deadline $t+k$. If the algorithm fails at this point, there are at least $k+1$ pages requested at time $t+k$, which means there is no feasible schedule.

Consider a fetch (of page p) that finishes at time t and suppose the algorithm did not fail yet. Greedy plans to fetch the first requested page with deadline t or greater that is not in its cache. The only case in which this fails is if there exists a page q with deadline smaller than $t+\frac{1}{s}$ which is not in the cache of Greedy. This page was already visible at time $t+\frac{1}{s}-\frac{k}{s-1}$. Greedy must have been loading pages throughout the interval $[t+\frac{1}{s}-\frac{k}{s-1}, t]$, loading $s(\frac{k}{s-1}-\frac{1}{s}) = \frac{sk}{s-1} - 1$ pages in this time. It was also loading a page immediately before time $t+\frac{1}{s}-\frac{k}{s-1}$, since otherwise it would have started to load q sooner. But this means that in the interval $[t+\frac{1}{s}-\frac{k}{s-1}, t+\frac{1}{s})$ there are $\frac{sk}{s-1}$ distinct pages requested (Greedy would not evict q at time $t+\frac{1}{s}-\frac{k}{s-1}$ anymore). However, by Lemma 9.7, there can be at most $k+\frac{k}{s-1} - 1 = \frac{sk-s+1}{s-1}$ pages requested in an interval of length less than $\frac{k}{s-1}$, a contradiction. □

We next show a matching lower bound, showing a tight relationship between the amount of lookahead and the amount of resource augmentation on the speed that is required for an online algorithm to provide feasible schedules for feasible inputs.

Theorem 9.2 *An online algorithm with a disk of speed s, or s parallel disks of speed 1, needs at least a lookahead of $k/(s-1)$ in order to be able to create feasible schedules for all feasible inputs.*

Proof Assume that the amount of lookahead is $\ell < k/(s-1)$ and consider the following instance. It consists of k pages requested at time k, followed by new distinct pages with deadlines at each time $k+i$ for $i = 1, \ldots, 2k$. At time $3k - \ell$, Greedy has at most k of the at least $2k$ pages with deadline no later than $3k - \ell$ in cache. We now add a request for k pages that Greedy does not have in its cache, but that were already requested, at time $3k+1$.

The optimal offline solution is the following: first load the k pages requested at time k in the interval $[0, k]$. In each successive interval of length 1 until time $3k$, load one page and evict one page that will not be requested again. It can only happen once that there is no such page in cache, namely if all k pages in cache are requested at time $3k+1$ (which is the only time at which requests are repeated). In that case, evict an arbitrary page, and reload it in the interval $[3k, 3k+1]$. In all cases, this produces a feasible solution.

In this instance, in the interval $(3k - \ell, 3k+1]$, there are deadlines for $k + \ell$ distinct pages. Loading all these pages takes at least $(k+\ell)/s$ time for Greedy. Thus we find as condition for ℓ such that Greedy might create a feasible schedule that $\ell \geq (k+\ell)/s$, or $\ell \geq k/(s-1)$. □

If we give the online algorithm parallel disks instead of a faster disk, we get slightly different results because now loading $k+\ell$ distinct pages takes at least $\lceil (k+\ell)/s \rceil$ time, so the required lookahead may be slightly larger depending on k, s and ℓ.

9.5 Conclusions

We have introduced a model for real-time prefetching and caching that is simple, seems to model practically relevant issues, and allows fast and simple algorithm with useful performance guarantees in offline and online settings. Although previous work from non-real-time models provides useful ideas for algorithms, the situation in the real-time setting is often different (e.g., wrt to I/O optimality of LFD or how to measure lookahead). Hence, given the importance of real-time applications, we expect that more work will be done on this subject in the future.

One interesting open question is the case of parallel disks. Although the *hard real-time* case we currently consider is very important since hard real-time constraints are present in many safety critical systems (e.g. avionics), we could also look at *soft real-time* where the applications remains viable when some requests are missed but we want to minimize the number of missed requests (or the sum of importance weights given for the missed requests).

Bibliography

[1] Marjan van den Akker, Han Hoogeveen, and Nodari Vakhania. Restarts can help in the on-line minimization of the maximum delivery time on a single machine. *Journal of Scheduling*, 3:333–341, 2000.

[2] S. Albers. On the influence of lookahead in competitive paging algorithms. *Algorithmica*, 18:283–305, 1997.

[3] Susanne Albers. Better bounds for online scheduling. *SIAM J. Comput.*, 29:459–473, 1999.

[4] Susanne Albers, Naveen Garg, and Stefano Leonardi. Minimizing stall time in single and parallel disk systems. *J. ACM*, 47(6):969–986, 2000.

[5] Noga Alon, Yossi Azar, János Csirik, Leah Epstein, Sergey V. Sevastianov, Arjen Vestjens, and Gerhard J. Woeginger. On-line and off-line approximation algorithms for vector covering problems. *Algorithmica*, 21:104–118, 1998.

[6] Noga Alon, Yossi Azar, Gerhard Woeginger, and Tal Yadid. Approximation schemes for scheduling. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 493–500, 1997.

[7] Baruch Awerbuch, Yossi Azar, Amos Fiat, Stefano Leonardi, and Adi Rosen. On-line competitive algorithms for call admission in optical networks. *Algorithmica*, 31(1):29–43, 2001. Also in J. Diaz, M. Serna, editors, *Algorithms - ESA '96, Proceedings Fourth Annual European Symposium*, volume 1136 of Lecture Notes in Computer Science, pages 431–444. Springer, 1996.

[8] Yossi Azar. On-line load balancing. In A. Fiat and Gerhard J. Woeginger, editors, *Online Algorithms - The State of the Art*, chapter 8, pages 178–195. Springer, 1998.

[9] Yossi Azar and Leah Epstein. On two dimensional packing. *Journal of Algorithms*, 25(2):290–310, 1997. Also in Proc. SWAT'96 pp. 321-332.

[10] Brenda S. Baker, Donna J. Brown, and Howard P. Katseff. A 5/4 algorithm for two-dimensional packing. *J. Algorithms*, 2:348–368, 1981.

[11] Brenda S. Baker, A. Robert Calderbank, Edward G. Coffman Jr., and Jeffrey C. Lagarias. Approximation algorithms for maximizing the number of squares packed into a rectangle. *SIAM Journal on Algebraic and Discrete Methods*, 4(3), 1983.

[12] Brenda S. Baker, Edward G. Coffman, and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9:846–855, 1980.

[13] Brenda S. Baker and J. S. Schwartz. Shelf algorithms for two-dimensional packing problems. *SIAM J. Comput.*, 12:508–525, 1983.

[14] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Dynamic speed scaling to manage energy and temperature. In *IEEE Syposium on Foundations of Computer Science*, pages 520 – 529, 2004.

[15] Nikhil Bansal, Andrea Lodi, and Maxim Sviridenko. A tale of two dimensional bin packing. In *Proc. 46th IEEE Symp. on Found. of Comp. Science*, 2005. To appear.

[16] Nikhil Bansal and Kirk Pruhs. Speed scaling to manage temperature. In *Symposium on Theoretical Aspects of Computer Science*, pages 460–471, 2005.

[17] Nikhil Bansal and Maxim Sviridenko. New approximability and inapproximability results for 2-dimensional packing. In *Proceedings of the 15th Annual Symposium on Discrete Algorithms*, pages 189–196. ACM/SIAM, 2004.

[18] Nikhil Bansal and Maxim Sviridenko. Two-dimensional bin packing with one dimensional resource augmentation. Manuscript, 2005.

[19] Yair Bartal, Amos Fiat, Howard Karloff, and Rakesh Vohra. New algorithms for an ancient scheduling problem. *J. Comput. Systems Sci.*, 51:359–366, 1995.

[20] Laszlo A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, 5:78–101, 1966.

[21] David Blitz, Andre van Vliet, and Gerhard J. Woeginger. Lower bounds on the asymptotic worst-case ratio of online bin packing algorithms. Unpublished manuscript, 1996.

[22] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[23] Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber. Competitive paging with locality of reference. *J. Comput. Systems Sci.*, 50:244–258, 1995.

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

[25] Donna J. Brown. A lower bound for on-line one-dimensional bin packing algorithms. Technical Report R-864, Coordinated Sci. Lab., Urbana, Illinois, 1979.

[26] Donna J. Brown, Brenda. S. Baker, and Howard. P. Katseff. Lower bounds for on-line two-dimensional packing algorithms. *Acta Informatica*, 18:207225, 1982.

[27] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *SIGMETRICS*, pages 188–197, 1995.

[28] Alberto Caprara. Packing 2-dimensional bins in harmony. In *Proc. 43th IEEE Symp. on Found. of Comp. Science*, pages 490–499, 2002.

[29] Alberto Caprara, Hans Kellerer, and Ulrich Pferschy. Approximation schemes for ordered vector packing problems. *Naval Research Logistics*, 92:58–69, 2003.

[30] Alberto Caprara, Andrea Lodi, and Michele Monaci. Fast approximation schemes for the two-stage, two-dimensional bin packing problem. *Mathematics of Operations Research*, 30:150–172, 2005.

[31] Alberto Caprara and Michele Monaci. On the 2-dimensional knapsack problem. *Operations Research Letters*, 32:5–14, 2004.

[32] Soumen Chakrabarti, Cynthia A. Phillips, A. S. Schulz, David B. Shmoys, C. Stein, and Joel Wein. Improved scheduling algorithms for minsum criteria. In *Proc. 23rd International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1099 of *Lecture Notes in Comput. Sci.*, pages 646–657. Springer, 1996.

[33] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 609–618, New York / Philadelphia, 1997. ACM / SIAM.

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

[35] Chandra Chekuri and Sanjeev Khanna. On multidimensional packing problems. *SIAM Journal on Computing*, 33(4):837–851, 2004.

[36] Jian-Jia Chen, Tei-Wei Kuo, and Hsueh-I Lu. Power-saving scheduling for weakly dynamic voltage scaling devices. In *Workshop on Algorithms and Data Structures*, 2005. To appear.

[37] Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9:91–103, 1988.

[38] Marek Chrobak and John Noga. LRU is better than FIFO. *Algorithmica*, 23:180–185, 1999.

[39] Fabián A. Chudak and David B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 581–590, 1997.

[40] Fan R. K. Chung, Michael R. Garey, and David S. Johnson. On packing two-dimensional bins. *SIAM J. on Algebraic and Discrete Methods*, 3:66–76, 1982.

[41] Edward G. Coffman, Michael R. Garey, and David S. Johnson. Dynamic bin packing. *SIAM J. Comput.*, 12:227–258, 1983.

[42] Edward G. Coffman, Michael R. Garey, and David S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation algorithms*. PWS Publishing Company, 1997.

[43] Edward G. Coffman, Michael R. Garey, David S. Johnson, and Robert E. Tarjan. Performance bounds for level oriented two-dimensional packing algorithms. *SIAM J. Comput.*, 9:808–826, 1980.

[44] Edward G. Coffman Jr., Peter J. Downey, and Peter M. Winkler. Packing rectangles in a strip. *Acta Inf.*, 38(10):673–693, 2002.

[45] Edward G. Coffman Jr. and Edgar N. Gilbert. Dynamic, first-fit packings in two or more dimensions. *Information and Control*, 61(1):1–14, 1984.

[46] Don Coppersmith and Prabhakar Raghavan. Multidimensional online bin packing: Algorithms and worst case analysis. *Oper. Res. Lett.*, 8:17–20, 1989.

[47] Jose Correa and Claire Kenyon. Approximation schemes for multidimensional packing. In *Proceedings of the 15th ACM/SIAM Symposium on Discrete Algorithms*, pages 179–188. ACM/SIAM, 2004.

[48] Jose R. Correa. Resource augmentation in two-dimensional packing with orthogonal rotations. *Operations Research Letters*. To appear.

[49] Janos Csirik. An online algorithm for variable-sized bin packing. *Acta Inform.*, 26:697–709, 1989.

[50] János Csirik, J. B. G. Frenk, and M. Labbe. Two dimensional rectangle packing: On line methods and results. *Discrete Appl. Math.*, 45:197–204, 1993.

[51] János Csirik and André van Vliet. An on-line algorithm for multidimensional bin packing. *Operations Research Letters*, 13(3):149–158, Apr 1993.

[52] Janos Csirik and Gerhard J. Woeginger. Shelf algorithms for on-line strip packing. *Inform. Process. Lett.*, 63:171–175, 1997.

[53] János Csirik and Gerhard J. Woeginger. On-line packing and covering problems. In *A. Fiat and G. J. Woeginger, editors,* Online Algorithms: The State of the Art, volume 1442 of *Lecture Notes in Computer Science*, pages 147–177. Springer-Verlag, 1998.

[54] János Csirik and Gerhard J. Woeginger. Resource augmentation for online bounded space bin packing. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 296–304, Jul 2000.

[55] Mauro Dell'Amico, Silvano Martello, and Daniele Vigo. A lower bound for the non-oriented two-dimensional bin packing problem. *Discrete Applied Mathematics*, 118:13–24, 2002.

[56] Leah Epstein. On-line variable sized covering. *Information and Computation*, 171(2): 294–305, 2001.

[57] Leah Epstein. On variable sized vector packing. *Acta Cybernetica*, 16:47–56, 2003.

[58] Leah Epstein. Two dimensional packing: the power of rotation. In *Proc. of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS'2003)*, pages 398–407, 2003.

[59] Leah Epstein and Meital Levy. Dynamic multi-dimensional bin packing. Manuscript, 2006.

[60] Leah Epstein and Rob van Stee. Lower bounds for on-line single-machine scheduling. In *Proc. 26th Symp. on Mathematical Foundations of Computer Science (MFCS)*, volume 2136 of *Lecture Notes in Comput. Sci.*, pages 338–350. Springer, 2001.

[61] Leah Epstein and Rob van Stee. Optimal online bounded space multidimensional packing. In *Proc. of 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 207–216. ACM/SIAM, 2004.

[62] Leah Epstein and Rob van Stee. Online square and cube packing. *Acta Informatica*, 41(9):595–606, 2005.

[63] Leah Epstein and Rob van Stee. Bounds for online bounded space hypercube packing. *Discrete optimization*, 2007. To appear.

[64] Thomas Erlebach. Private communication, 2005.

[65] Ö. Ertug, M. Kallahalla, and P. J. Varman. Real time parallel disk scheduling for vbr video servers. In *Proc. of Fifth Intl. Conf. On Computer Science and Informatics (CSI'00)*, Chennai, India, 2000.

[66] Anja Feldmann, Jiří Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. *Theoret. Comput. Sci.*, 130:49–72, 1994.

[67] Wenceslas Fernandez de la Vega and George S. Lueker. Bin packing can be solved within $1 + \varepsilon$ in linear time. *Combinatorica*, 1:349–355, 1981.

[68] Carlos E. Ferreira, Flavio K. Miyazawa, and Yoshiko Wakabayashi. Packing squares into squares. *Pesquisa Operacional*, 19(2):223–237, 1999.

[69] Amos Fiat and M. Mendel. Truly online paging with locality of reference. In *Proc. 38th Symp. Foundations of Computer Science (FOCS)*, pages 326–335. IEEE, 1997.

[70] Amos Fiat and Jared Saia. Censorship resistant Peer-to-Peer content addressable networks. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Algorithms (SODA-02)*, pages 94–103, 2002.

[71] Rudolph Fleischer and Michaela Wahl. On-line scheduling revisited. *J. Sched.*, 3:343–353, 2000.

[72] Erich Friedman. Packing unit squares in squares: a survey and new results. *Electronic J. Comb.*, 7, 2000.

[73] D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM J. Comput.*, 15:222–230, 1986.

[74] Satoshi Fujita and Takeshi Hada. Two-dimensional on-line bin packing problem with rotatable items. *Theoretical Computer Science*, 289(2):939–952, 2002.

[75] Gabor Galambos. A 1.6 lower bound for the two-dimensional online rectangle bin packing. *Acta Cybernet.*, 10:21–24, 1991.

[76] Gabor Galambos, H. Kellerer, and Gerhard J. Woeginger. A lower bound for online vector packing algorithms. *Acta Cybernet.*, 10:23–34, 1994.

[77] Gabor Galambos and Andre van Vliet. Lower bounds for 1-, 2-, and 3-dimensional online bin packing algorithms. *Computing*, 52:281–297, 1994.

[78] Michael R. Garey, Ronald L. Graham, David S. Johnson, and A. C. C. Yao. Resource constrained scheduling as generalized bin packing. *J. Combin. Theory Ser. A*, 21:257–298, 1976.

[79] Michael R. Garey, Ronald L. Graham, and Jeffrey D. Ullman. Worst-case analysis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, pages 143–150. ACM, 1972.

[80] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the theory of of NP-Completeness*. Freeman and Company, San Francisco, 1979.

[81] T. Gonzalez, O. H. Ibarra, and S. Sahni. Bounds for LPT Schedules on Uniform Processors. *SIAM Journal on Computing*, 6(1):155–166, 1977.

[82] Todd Gormley, Nick Reingold, Eric Torng, and Jeffery Westbrook. Generating adversaries for request-answer games. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 564–565. ACM-SIAM, 2000.

[83] Ronald L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Techical Journal*, 45:1563–1581, 1966.

[84] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17:263–269, 1969.

[85] Ronald L. Graham, Eugene Lawler, Jan Karel Lenstra, and Alexander H. G. Rinnooy Kan. Optimization and approximation in deterministic scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[86] Flavius Gruian and Krzysztof Kuchcinski. Lenes: Task-scheduling for low-energy systems using variable voltage processors. In *Asia South Pacific - Design Automation Conference*, pages 449–455, 2001.

[87] Xin Han, Kazuo Iwama, and Guochuan Zhang. Online removable square packing. In *Proceedings of the 3rd Workshop on Approximation and Online Algorithms (WAOA 2005)*, Lecture Notes in Computer Science, pages 216–229. Springer, 2006.

[88] Xin Han, Deshi Ye, and Yong Zhou. Improved online hypercube packing. In *Proceedings of the 4th Workshop on Approximation and Online Algorithms (WAOA 2006)*. To appear.

[89] Han Hoogeveen, Chris N. Potts, and Gerhard J. Woeginger. On-line scheduling on a single machine: Maximizing the number of early jobs. *Oper. Res. Lett.*, 27:193–196, 2000.

[90] J. A. Hoogeveen and Arjen P. A. Vestjens. Optimal on-line algorithms for single-machine scheduling. In *Proc. 5th Conf. Integer Programming and Combinatorial Optimization (IPCO)*, volume 1084 of *Lecture Notes in Comput. Sci.*, pages 404–414. Springer, 1996.

[91] David A. Hutchinson, Peter Sanders, and Jeffrey Scott Vitter. Duality between prefetching and queued writing with parallel disks. In *Algorithms - ESA 2001, 9th Annual European Symposium*, pages 62–73, 2001.

[92] Csanád Imreh. Online strip packing with modifiable boxes. *Operation Research Letters*, 66:79–86, 2001.

[93] Sandy Irani and Kirk Pruhs. Algorithmic problems in power management. *SIGACT News*, 2005.

[94] Klaus Jansen and Rob van Stee. On strip packing with rotations. In *Proceedings of the 37th ACM Symposium on Theory of Computing (STOC 2005)*, pages 755–761. ACM, 2005.

[95] Klaus Jansen and Guochuan Zhang. Maximizing the number of packed rectangles. In *Proc. of the 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, pages 362–371, 2004.

[96] Klaus Jansen and Guochuan Zhang. On rectangle packing: maximizing benefits. In *Proceedings of the 15th Annual Symposium on Discrete Algorithms (SODA'04)*, pages 204–213, 2004.

[97] Janusz Januszewski and Marek Lassak. Online packing sequences of cubes in the unit cube. *Geometriae Dedicata*, 67:285–293, 1997.

[98] David S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, MIT, Cambridge, MA, 1973.

[99] David S. Johnson. Fast algorithms for bin packing. *J. Comput. Systems Sci.*, 8:272–314, 1974.

[100] M. Kallahalla and P. J. Varman. Optimal read-once parallel disk scheduling. In *6th Workshop on Input/Output in Parallel and Distributed Systems*, pages 68–77, 1999.

[101] M. Kallahalla and P. J. Varman. Optimal prefetching and caching for parallel I/O systems. In *13th Symposium on Parallel Algorithms and Architectures*, pages 219–228, 2001.

[102] D. R. Karger, Steven J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. *J. Algorithms*, 20:400–430, 1996.

[103] Anna Karlin, Mark Manasse, Larry Rudolph, and Daniel Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[104] Narendra Karmarkar and Richard M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 312–320, 1982.

[105] Hans Kellerer and Vladimir Kotov. An approximation algorithm with absolute worst-case performance ratio 2 for two-dimensional vector packing. *Oper. Res. Lett.*, 31(1):35–41, 2003.

[106] Claire Kenyon. Personal communication.

[107] Claire Kenyon and Eric Rémila. A near optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research*, 25(4):645–656, 2000.

[108] Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *SIAM J. Comput.*, 29(4):1051–1082, 2000.

[109] Yoshiharu Kohayakawa, Flavio K. Miyazawa, Prabhakar Raghavan, and Yoshiko Wakabayashi. Multidimensional cube packing. *Algorithmica*, 40(3):173–187, 2004.

[110] L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. *IBM J. Res. Dev.*, 21:443–448, 1977.

[111] P. Krishnan and Jeffrey Scott Vitter. Optimal prediction for prefetching in the worst case. *SIAM J. Comput.*, 27(6):1617–1636, 1998.

[112] Piotr Krysta, Peter Sanders, and Berthold Vöcking. Scheduling and traffic allocation for tasks with bounded splittability. In *Proc. of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS 2003)*, pages 500–510, 2003.

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

[114] C. C. Lee and D. T. Lee. A simple online bin packing algorithm. *J. ACM*, 32:562–572, 1985.

[115] Joseph Y.-T. Leung, Tommy W. Tam, C. S. Wong, Gilbert H. Young, and Francis Y. L. Chin. Packing squares into a square. *Journal on Parallel and Distributed Computing*, 10:271–275, 1990.

[116] K. Li and K. H. Cheng. Generalized First-Fit algorithms in two and three dimensions. *Int. J. on Found. Comput. Sci.*, 2:131–150, 1990.

[117] K. Li and K. H. Cheng. Heuristic algorithms for online packing in three dimensions. *J. Algorithms*, 13:589–605, 1992.

[118] Keqin Li and Kam-Hoi Cheng. A generalized harmonic algorithm for on-line multidimensional bin packing. Technical Report UH-CS-90-2, University of Houston, January 1990.

[119] Keqin Li and Kam-Hoi Cheng. On three-dimensional packing. *SIAM Journal on Computing*, 19(5):847–867, 1990.

[120] Keqin Li and Kam-Hoi Cheng. Static job scheduling in partitionable mesh connected systems. *Journal on Parallel and Distributed Computing*, 10:152–159, 1990.

[121] Minming Li, Becky Jie Liu, and Frances F. Yao. Min-energy voltage allocation for tree-structured tasks. In *11th International Computing and Combinatorics Conference (COCOON 2005)*, 2005. To appear.

[122] R. Li and L. Shi. An on-line algorithm for some uniform processor scheduling. *SIAM Journal on Computing*, 27(2):414–422, 1998.

[123] F. M. Liang. A lower bound for online bin packing. *Inform. Process. Lett.*, 10:76–79, 1980.

[124] Jane W. S. Liu and C. L. Liu. Bounds on scheduling algorithms for heterogeneous computing systems. In Jack L. Rosenfeld, editor, *Proceedings of IFIP Congress 74*, volume 74 of *Information Processing*, pages 349–353, 1974.

[125] Jiong Luo and Niraj K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic task graphs in distributed real-time embedded systems. In *International Conference on Computer Aided Design*, pages 357–364, 2000.

[126] A. Meir and L. Moser. On packing of squares and cubes. *J. Combin. Theory*, 5:126–134, 1968.

[127] Ramesh Mishra, Namrata Rastogi, Dakai Zhu, Daniel Moss, and Rami G. Melhem. Energy aware scheduling for distributed real-time systems. In *International Parallel and Distributed Processing Symposium*, page 21, 2003.

[128] Flavio Keidi Miyazawa and Yoshiko Wakabayashi. An algorithm for the three-dimensional packing problem with asymptotic performance analysis. *Algorithmica*, 18(1):122–144, 1997.

[129] Flavio Keidi Miyazawa and Yoshiko Wakabayashi. Approximation algorithms for the orthogonal z-oriented 3-d packing problem. *SIAM Journal on Computing*, 29(3):1008–1029, 1999.

[130] Flavio Keidi Miyazawa and Yoshiko Wakabayashi. Cube packing. *Theoretical Computer Science*, 297(1-3):355–366, 2003.

[131] Flavio Keidi Miyazawa and Yoshiko Wakabayashi. Packing problems with orthogonal rotations. In M. Farach-Colton, editor, *Theoretical Informatics, 6th Latin American Symposium*, number 2976 in Lecture Notes in Computer Science, pages 359–368, 2004.

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

[133] E. Naroska and U. Schwiegelshohn. On an on-line scheduling problem for parallel jobs. *Information Processing Letters*, 81(6):297–304, 2002.

[134] Pavel Novotný. On packing of squares into a rectangle. *Archivum Mathematicum*, 32:75–83, 1996.

[135] Konstantinos Panagiotou and Alexander Souza. On adequate performance measures for paging. In *STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 487–496, New York, NY, USA, 2006. ACM Press.

[136] C.A. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. In *Proceedings of the 4th Workshop on Algorithms and Data Structures (WADS'95)*, volume 955 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 1995.

[137] Kirk Pruhs, Patchrawat Uthaisombut, and Gerhard Woeginger. Getting the best response for your erg. In *Scandanavian Workshop on Algorithms and Theory*, pages 14–25, 2004.

[138] Ingo Schiermeyer. Reverse-fit: a 2-optimal algorithm for packing rectangles. In *Algorithms - ESA '94, Proceedings Second Annual European Symposium*, pages 290–299, 1994.

[139] Steve S. Seiden. An optimal online algorithm for bounded space variable-sized bin packing. *SIAM Journal on Discrete Mathematics*, 14(4):458–470, 2001.

[140] Steve S. Seiden. On the online bin packing problem. *Journal of the ACM*, 49(5):640–671, 2002.

[141] Steve S. Seiden and Rob van Stee. New bounds for multi-dimensional packing. *Algorithmica*, 36(3):261–293, 2003.

[142] Jiří Sgall. *On-line scheduling on parallel machines*. PhD thesis, Technical Report CMU-CS-94-144, Carnegie-Mellon University, Pittsburgh, PA, U.S.A., 1994.

[143] Jiří Sgall. A lower bound for randomized on-line multiprocessor scheduling. *Information Processing Letters*, 63:51–55, 1997.

[144] Hadas Shachnai and Tami Tamir. Multiprocessor scheduling with machine allotment and parallelism constraints. *Algorithmica*, 32(4):651–678, 2002.

[145] David B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on line. *SIAM J. on Computing*, 24:1313–1331, 1995.

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

[147] Daniel D. K. D. B. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Inform. Process. Lett.*, 10:37–40, 1980.

[148] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[149] Rob van Stee and Johannes A. La Poutré. Running a job on a collection of dynamic machines, with on-line restarts. *Acta Informatica*, 37(10):727–742, 2001.

[150] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, April 1997.

[151] Leen Stougie. Unpublished manuscript, 1995.

[152] Leen Stougie and Arjen P.A. Vestjens. Randomized on-line scheduling: How low can't you go? Unpublished manuscript.

[153] Jeffrey D. Ullman. The performance of a memory allocation algorithm. Technical Report 100, Princeton University, Princeton, NJ, 1971.

[154] Arjen P. A. Vestjens. *On-line Machine Scheduling*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1997.

[155] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. *J. ACM*, 43(5):771–793, 1996.

[156] André van Vliet. An improved lower bound for online bin packing algorithms. *Inform. Process. Lett.*, 43:277–284, 1992.

[157] André van Vliet. *Lower and upper bounds for online bin packing and scheduling heuristics*. PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1995.

[158] Gerhard J. Woeginger. There is no asymptotic PTAS for two-dimensional vector packing. *Inf. Process. Lett.*, 64(6):293–297, 1997.

[159] A. C. C. Yao. New algorithms for bin packing. *J. ACM*, 27:207–227, 1980.

[160] F. Frances Yao, Alan J. Demers, and Scott Shenker. A scheduling model for reduced cpu energy. In *IEEE Syposium on Foundations of Computer Science (FOCS 1995)*, pages 374–382, 1995.

[161] Han-Saem Yun and Jihong Kim. On energy-optimal voltage scheduling for fixed priority hard real-time systems. *ACM Transactions on Embedded Computing Systems*, 2(3):393–430, 2003.

[162] Guochuan Zhang. A 3-approximation algorithm for two-dimensional bin packing. *Operations Research Letters*, 33(2):121–126, 2005.

[163] Yumin Zhang, Xiaobo Hu, and Danny Z. Chen. Task scheduling and voltage selection for energy minimization. In *Design Automation Conference*, pages 183–188, 2002.