Approximation Algorithms for Modern Multi-Processor Scheduling Problems

THÈSE N^{O} 5561 (2012)

PRÉSENTÉE LE 28 NOVEMBRE 2012

À LA FACULTÉ DES SCIENCES DE BASE

CHAIRE D'OPTIMISATION DISCRÈTE

PROGRAMME DOCTORAL EN MATHÉMATIQUES

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Martin NIEMEIER

acceptée sur proposition du jury:

Prof. K. Hess Bellwald, présidente du jury
Prof. F. Eisenbrand, directeur de thèse
Prof. S. Baruah, rapporteur
Prof. N. Megow, rapporteur
Prof. O. N. A. Svensson, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

If people do not believe that mathematics is simple,
it is only because they do not realize how complicated life is.
— John von Neumann (1903 – 1957)

Acknowledgements

First and foremost I would like to express my deepest gratitude to my advisor Prof. Dr. Friedrich Eisenbrand. With his excellent lectures on linear and combinatorial optimization in Paderborn he sparked my interest in this research area in the first place. Later, when I joined his Discrete Optimization Group at EPFL, he was a great advisor. He provided guidance that is invaluable to me, but at the same time granted freedom to explore the world of combinatorial optimization and discrete mathematics to develop my research interests. I also want to thank him for the enjoyable and fruitful collaboration, as he his a co-author of many of my publications, and for his constant encouragement, especially during the time of writing this thesis.

I am also indebted to my other co-authors Sanjoy Baruah, Nicolai Hähnle, Raju Mattikalli, Arnold Nordsieck, Martin Skutella, José Verschae and Andreas Wiese, without whom this work would not have been possible. I am particularly grateful to Sanjoy, who introduced me to and motivated most research questions regarding scheduling problems that I worked on during my time as a PhD-candidate. He also established the contact to our industrial partner. Special thanks also goes to Andreas for many great and fruitful discussions on several topics of my thesis during his visits to Lausanne and my visits to Berlin, and to my office-mate Nicolai for a lot of insightful and delightful discussions on my other research topics (that do not appear in this thesis) and beyond.

I would like to thank my friends and/or former colleagues from DISOPT (aforely mentioned Fritz and Nicolai, but also Adrian, Andreas, Carsten, Dave, Genna, Hans, Jarek, Jocelyne, Laura, Mădălina, Marco, Nicolas, Rico and Thomas) and DCG (Andres, Andrew, Balázs, Dömötör, Fabrizio, Filip, Gabriel, János, Nabil, Padmini, Rado, Rom). All of you contributed not only to a very stimulating and vivid working atmosphere during my time at EPFL, but also to a fantastic and exceptional extraworkular environment. Special thanks goes to "Gruppenbeauftragter" Adrian for his kind help and assistance in so many things I won't even start enumerating them.

I would also like to thank my jury members Sanjoy Baruah, Fritz Eisenbrand, Nicole Megow and Ola Svensson for their careful reading of this thesis and their valuable comments, most of which have been implemented in this final version. In particular I would like to thank Nicole for helpful discussions on online scheduling during the final stages of my thesis writing.

Thanks to Adrian Bock and Christian Ikenmeyer for reading parts of the manuscript and their helpful comments.

Finally, I would like to express my deepest gratitude to my parents for their loving and ongoing support in all parts of my life.

Berlin, October 2012 M. N.

Abstract

This thesis is devoted to the design and analysis of algorithms for scheduling problems. These problems are ubiquitous in the modern world. Examples include the optimization of local transportation, managing access to concurrent resources like runways at airports and efficient execution of computing tasks on server systems. Problem instances that appear in the real world often are so large and complex that it is not possible to solve them “by hand”. This rises the need for strong algorithmic approaches, which motivates our focus of study.

In this work we consider two types of scheduling problems which gained in importance due to recent technological advances. The first problem comes from the avionics industry and deals with scheduling periodically recurring tasks in a parallel computer network on a plane: Each task comes with a period p and execution time c, and needs to use a processor exclusively for c time units every p time units. The scheduling problem is to assign starting offsets for the first execution of the tasks so that no collision occurs. The second problem is a scheduling problem that arises in highly parallelized processing environments with a shared common resource, e.g., modern multi-core computer architectures. In addition to classical makespan minimization problems such as scheduling on identical machines, each job has an additional resource constraint. The scheduler must ensure that at no time, the accumulated requirement of all active jobs at that time exceeds a given limit.

For both types of problems we study their algorithmic complexity in a mathematical, rigorous way by designing approximation algorithms and establishing inapproximability results. We thereby give a characterization of the approximation landscape of these problems.

We also consider a more practical perspective: For an engineer from the industry, a rigorous proof that an algorithm finds a solution of certain guaranteed quality for all possible kinds of problem instances is usually not that relevant. It is rather of interest to find “good enough” or even optimal solutions for particular instances that actually appear in the real world in “reasonable” time. We show that structural insights gained in the more theoretical process of designing approximation algorithms can be highly beneficial also for obtaining practical results. In particular, we develop integer programming formulations for the avionics problem based on structural properties revealed in the design of approximation algorithms. These formulations lead to strong tools that, for the first time, enable to algorithmically solve real-world instances from our industrial partner.

Keywords: scheduling; real-time scheduling; periodic maintenance problem; resource constrained scheduling; approximation algorithms; offline algorithms; online algorithms; integer programming; mathematical modeling

Zusammenfassung

In dieser Arbeit befassen wir uns mit dem Entwurf und der Analyse von Algorithmen zum Lösen von Schedulingproblemen. Derartige Probleme sind allgegenwärtig in der modernen Welt. Beispielsweise gehören die Erstellung von Fahrplänen für den öffentlichen Nahverkehr oder das effiziente Ausführen von Berechnungen in Computern oder Rechenzentren allesamt zur Klasse der Schedulingprobleme. Die dabei auftretenden Probleminstanzen sind in der Regel derart komplex, dass es nicht möglich ist, sie "von Hand" zu lösen. Deshalb ist es notwendig, algorithmische Methoden zum Lösen dieser Probleme zu entwickeln.

Diese Arbeit beschäftigt sich mit dem Entwurf solcher Algorithmen für zwei Arten von Schedulingproblemen, die in letzter Zeit aufgrund technologischen Fortschritts an Bedeutung gewonnen haben. Die erste Problemklasse stammt aus der Luftfahrtindustrie. Hier geht es darum, periodisch auftretende Tasks in einem parallelen Rechnernetzwerk eines Flugzeugs auszuführen: Die Aufgabe des Schedulingalgorithmus ist es, jeder Task einen Start-Offset zuzuordnen, so dass Kollisionen vermieden werden. Die zweite Klasse von Schedulingproblemen tritt in hoch parallelisierten Rechnerumgebungen auf, wie beispielsweise in modernen Mehrkern-Computerarchitekturen. Zusätzlich zu klassischen Makespan-Minimierungsproblemen besitzt jeder Job einen gewissen Ressourcenbedarf. Der Scheduler muss sicherstellen, dass der gesamte Ressourcenbedarf der Jobs, welche zu einem bestimmten Zeitpunkt aktiv sind, ein vorgegebenes Limit nicht überschreitet.

Wir untersuchen die algorithmische Komplexität beider Problemklassen. Dazu entwickeln wir Approximationsalgorithmen und beweisen Inapproximierbarkeitsresultate. Auf diese Weise liefern wir eine Charakterisierung der Approximierbarkeit der Probleme.

Für einen Ingenieur in der Industrie ist ein mathematisch korrekter Beweis, dass ein bestimmter Algorithmus immer, für jede erdenkliche Probleminstanz, eine Lösung gewisser Güte erreicht, von nachrangiger Bedeutung. Vielmehr ist er daran interessiert, für seine konkret auftretenden Probleminstanzen eine gute oder sogar optimale Lösung zu finden. Wir werden zeigen, wie Wissen über die Struktur der Probleme, welches wir bei der theoretischen Analyse erworben haben, auch für diese praktischere Sicht hilft. Für die Probleme aus der Luftfahrtindustrie entwickeln wir ganzzahlige Programme, welche auf diesen strukturellen Eigenschaften des Problems basieren. Auf diese Weise erhalten wir mächtige Werkzeuge, um erstmalig die Probleminstanzen unseres Industriepartners zu lösen.

Schlagworte: Scheduling; Echtzeitscheduling; Periodic Maintenance Problem; Resource Constrained Scheduling; Approximationsalgorithmen; Offlinealgorithmen; Onlinealgorithmen; Ganzzahlige Programmierung; Mathematische Modellierung

Contents

Acknowledgements v

Abstract (English/Deutsch) vii

List of Figures xii

List of Tables xiii

1 Introduction 1
1.1 Real-Time Avionics Optimization 2
1.2 Resource Constrained Scheduling 3
1.3 Organization 5
1.4 Sources 5
1.5 Prerequisites 5

2 Real-Time Avionics Optimization 7
2.1 Introduction 7
2.1.1 Related Work 8
2.1.2 Our Contribution 8
2.1.3 Outline 9
2.2 The General Case 10
2.2.1 Prerequisites from Modular Arithmetic 10
2.2.2 General Concepts 11
2.2.3 Popular heuristics fail for periodic maintenance 13
2.2.4 Inapproximability 13
2.3 The Harmonic Case 18
2.3.1 Bin Trees 18
2.3.2 First-Fit: A 2-Approximation Algorithm 20
2.3.3 Inapproximablity 24
2.4 Further Research Directions 27

3 Real-time Avionics Optimization in Practice 29
3.1 Introduction 29
3.1.1 Problem Definition 30
3.1.2 Related Work 30

3.1.3 Our Contribution 31
3.1.4 Outline 32
3.2 General IP-Formulations 32
3.2.1 Time-Indexed-Formulation 32
3.2.2 Congruence-Formulation 33
3.3 Harmonic IP-Formulation 34
3.3.1 Structural Insights 34
3.3.2 Bin-Formulation 35
3.4 Extended Constraints 37
3.5 Computational Results 38

4 Resource Constrained Scheduling: Computing Schedules Offline **43**
4.1 Introduction 43
4.1.1 The Model and Basic Notation 44
4.1.2 Related Work 45
4.1.3 Our Contribution 46
4.1.4 Outline 48
4.2 Scheduling on Identical Machines 49
4.2.1 List Scheduler: A 3-Approximation Algorithm 49
4.2.2 The 2-Approximation Scheme 54
4.2.3 Polynomial Time Approximation Schemes for Special Cases 64
4.2.4 A Binary Search Framework 67
4.2.5 Inapproximability 69
4.3 Resource Constrained Scheduling on Unrelated Machines 69
4.3.1 Scheduling on Unrelated Parallel Machines with Costs 70
4.3.2 A 4-Approximation Algorithm 71
4.3.3 Improving the Approximation Ratio to 3.75 76
4.4 Further Research Questions 79

5 Resource Constrained Scheduling: Computing Schedules Online **81**
5.1 Introduction 81
5.1.1 The Model 81
5.1.2 Related Work 83
5.1.3 Our Contribution and Outline 84
5.2 An Online Algorithm for Resource Constrained Scheduling 85
5.3 Lower Bounds on the Competitive Ratio 89
5.4 Further Research Questions 90

6 Summary and Conclusion **91**

Bibliography **97**

Curriculum Vitae **99**

List of Figures

1.1 An example of an infeasible and a feasible schedule for the avionics scheduling problem. 3

2.1 Reduction of the COLORING problem to the machine minimization problem. . . 14
2.2 A schedule for a single machine. The gray jobs belong to the task with period length q_1, the striped jobs to tasks with period length $q_2 = 3 \cdot q_1$, and the dotted jobs to tasks with period length $q_3 = 6 \cdot q_1$. 18
2.3 The full bin tree corresponding to the schedule in Figure 2.2. 19
2.4 The compact form of the bin tree in Figure 2.3. 19
2.5 The situation of Lemma 2.3.3: Every group of machines but the last has a witness. The last two machines of the last group provide a witness for the rest. 22
2.6 Possible situation in the proof of Theorem 2.3.1: Group $\mathscr{M}_2$ does not have a witness. The dark tasks form the set $\overline{\mathscr{T}}$. Together with the white tasks, they form the set $\mathscr{T}'$. The striped tasks form the set $\mathscr{T}''$. 23

3.1 A schedule for a single machine and a schedule for the same tasks which is in bin structure. The gray jobs belong to tasks with period length q_1, the striped jobs to tasks with period length $q_2 = 3 \cdot q_1$, and the checkered jobs to tasks with period length $q_3 = 6 \cdot q_1$. 34
3.2 The bin tree corresponding to the schedule in Figure 3.1. 35

4.1 An illustration of the proof of Lemma 4.2.9. 54
4.2 A conceptual schedule computed by the $(2+\epsilon)$-approximation algorithm. The striped jobs are the ones assigned in step 1a. The hatched ones are those assigned in step 1b. The dotted ones are assigned in step 2a, and the dark ones are added in step 2b. 55
4.3 A feasible schedule scaled by a factor of $1+\epsilon$. The inner black boxes within the slots depict the processing time of the jobs assigned to the slots. The difference between slot-width and job-width illustrates the slack. 56
4.4 Illustration of the construction of a non-relaxed schedule from an LP-solution. 62

List of Tables

2.1 Overview of our results regarding the periodic maintenance minimization problem. 9

3.1 Computational results for the purely random instances in the non-harmonic case. 40
3.2 Computational results for the purely random instances in the harmonic case (basic PMP). 41
3.3 Computational results for the RWP instances (harmonic case) for the basic PMP. 41
3.4 Computational results for the RWP instances (harmonic case) for the extended PMP. 42

4.1 The complexity landscape of the resource constrained scheduling problem. . . 48

1 Introduction

Scheduling problems are ubiquitous in the modern world. Timetables for trains and buses in public transportation networks need to be created in such a way that passengers can travel between places quickly and without unnecessary waiting times. Flights at airports need to be scheduled so that limited resources of airports (e.g., runways) are used efficiently and the throughput (number of passengers/planes) is maximized. The phases of traffic lights need to be coordinated so that average traveling times are minimized. Jobs and tasks need to be processed on computers with parallel processing cores. All these problems are scheduling problems. Often, problem instances that appear in the real world are so large and complex that it is not possible to solve them “by hand”. This rises the need for strong algorithmic approaches to these scheduling problems. This thesis is devoted to the study of such algorithms for several scheduling problems which gained in importance due to recent technological advances. These problems are typically formulated as *optimization* problems: Find a feasible scheduling solution minimizing the usage of certain resources (machines, processors, workers, etc.). What characterizes a good algorithm in this context? Clearly, an algorithm that computes an optimal solution (i.e., a solution that minimizes the resources used) efficiently is highly preferable. Unfortunately, this perfect goal is usually not achievable. Under standard complexity theoretical assumptions, these problems are hard and there is no hope for an algorithm that always finds an *optimal* solution for these problems efficiently. Instead we resort to developing algorithms that always find a “near optimal” solution efficiently, even for the worst problem instances. These kinds of algorithms are called *approximation algorithms.*

From a more practical perspective, the characteristics of a good algorithm might differ. For an engineer from the industry, a rigorous proof that an algorithm finds a solution of certain guaranteed quality for *all* possible kinds of problem instances is usually not that relevant. It is rather of interest to find “good enough” or even optimal solutions for *particular* instances that actually matter for the concrete applications, in ”reasonable“ time. Although the main focus of this thesis is on approximation algorithms, we also discuss the latter kind of more practically oriented algorithms. In particular, we will see examples where structural insights

gained in the theoretical and rigorous process of designing approximation algorithms can be extremely beneficial also for obtaining the latter kind of results.

In this thesis we study two types of scheduling problems. The first one is a real-time scheduling problem from the avionics industry. The second one is a scheduling problem that arises in highly parallelized environments with a shared common resource, e.g., modern computer architectures. We first introduce both problem types in more detail and then discuss the outline of this document.

1.1 Real-Time Avionics Optimization

In real-time scheduling problems, one is typically given a set of tasks and has to ensure that all tasks can be processed in time without violating their deadlines. Often these tasks are of periodic nature: they emit jobs periodically, and each of these jobs needs to finish execution before its (relative) deadline. Real-time scheduling is a very broad and active area of research. We refer to part four of a book by Leung et al. [LKA04] for an extensive introduction. Some variants are well understood, while fundamental questions still remain open. For more details, see also a recent survey paper by Davis and Burns [DB11].

Here we are concerned with a special type of real-time scheduling problem. The motivation for this research comes from a real-word combinatorial optimization problem that was communicated to us by our industrial partner, a major avionics company. Modern airplanes use sophisticated computer systems for steering and controlling the plane, communicating with the air-traffic controller, and many other tasks. Also fly-by-wire control systems or autopilots would not be possible without the help of sophisticated on-board computing networks. In order to guarantee flight safety, the network must operate according to its specifications at all times. While for a home computer a small discrepancy between expected and actual behavior is perfectly acceptable, for a network that controls an aircraft it is not. Even a slight delay in the execution of a program might result in serious and even fatal problems. For that reason, the engineers rely on a static scheduling policy such that preemption and dynamic effects are avoided.

The model that is used in this context is as follows. One is given a set $\mathcal{T}$ of *tasks*. Each task $\tau \in \mathcal{T}$ is characterized by its *execution time* $c(\tau) \in \mathbb{N}$ and *period* $p(\tau) \in \mathbb{N}$. The goal is to assign the tasks to identical machines and to compute offsets $a(\tau) \in \mathbb{N}_0$ such that no collision occurs. A task $\tau \in \mathcal{T}$ generates one *job* with execution time $c(\tau)$ at every time unit $a(\tau) + p(\tau) \cdot k$ for all $k \in \mathbb{N}_0$. Each job needs to be processed immediately and non-preemptively after its generation on the task's machine. A collision occurs if two jobs are simultaneously active on one machine. See Figure 1.1 for an illustration of the problem on one machine: The upper picture shows an infeasible choice of offsets, a collision occurs. The lower picture shows a feasible choice of offsets for the same instance.

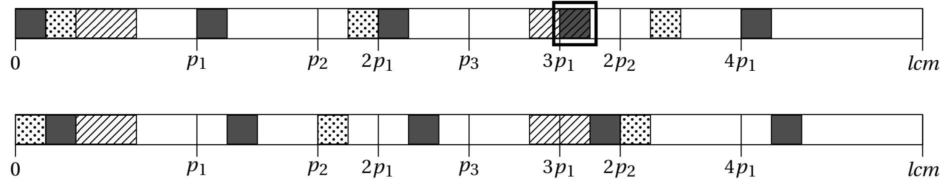

Figure 1.1: An example of an infeasible and a feasible schedule for the avionics scheduling problem.

One advantage of this model is that a scheduling algorithm following this model is very simple to implement. This allows to use code verification techniques to certify correctness of the scheduling algorithm, something that is crucial in safety-critical applications. One disadvantage of the model is that the available resources (machines) might not be used efficiently, i.e., depending on the structure of a problem instance, it is possible that even in an optimal solution, many machines are idle most of the time. However for the applications in the avionics industry, the advantage of simplicity out-weights the loss in efficiency.

In this work we analyze the algorithmic complexity of this problem. It turns out that the problem is quite intractable. We show that even simple problems like the following cannot be solved efficiently even if all execution times are 1. Assume that one is given a set of tasks $\mathcal{T}$ and feasible offsets for these tasks for one machine. Now consider another task τ and decide whether there is an offset $a(\tau)$ so that task τ can be added to the existing schedule without creating a collision. Even harder is the optimization variant, where a given task set should be distributed to a minimum number of machines in a feasible way: there is no efficient approximation algorithm with “nontrivial” approximation guarantee. It turns out however that instances from the industry usually have a special structure: The periods are *harmonic*, meaning that for any two periods of two tasks from the instance, one is an integer multiple of the other. For this special case, we develop special data structures that allow us to obtain “good” approximation algorithms. The structural insights also help to develop novel modeling methods. They enable for the first time to compute optimal solutions for real-world instances from the industry. For a more formal overview of our results regarding the problem, we refer to Section 2.1.2 and Section 3.1.3.

1.2 Resource Constrained Scheduling

Modern computer architectures allow for a high degree of parallelization: Multi-core CPUs and GPUs provide dozens (in the case of GPUs even hundreds) of parallel processing units. Usually these processing units share common resources, e.g., the cache of a multi-core processor. Hence, in order to utilize these systems efficiently, a scheduler must take into account the limited resources when executing jobs on the parallel machines. This poses new algorithmic challenges in designing efficient schedulers that respect these resource limitations.

We study the following model for the problem, which is called the *resource constrained scheduling problem*. An instance consists of a set of n jobs and m machines. Each job comes with a processing time and a *resource requirement*. The former specifies for how long a job needs to be executed to be completed. The latter specifies the amount of the common shared resource the job requires while it is active in order to work properly. The goal is to compute a *schedule*, i.e., a map that assigns to each job a time-interval of sufficient length on a machine. To model the shared resource, an "orthogonal" constraint is introduced: for a time-index t, the *resource usage* at t is the sum of resource requirements of jobs active at time t. The *resource constraint* requires that the resource usage never exceeds a given limit. A feasible schedule for our scheduling problem must respect the resource constraint. The goal is to compute a schedule of minimum makespan, i.e., a schedule where the last finishing time of a job is minimized.

The resource constrained scheduling problem has versatile applications, not only to model shared memory on parallel processors as previously mentioned. For example it can also be used to assign a limited amount of workers to different maintenance tasks, or to cope with the limited energy available for computing tasks. Next to the practical applications, the problem is also quite intriguing from a theoretical perspective. It can be seen as a hybrid of two classical problems from two different domains. On the one hand, one obtains classical minimum makespan scheduling problems if all resource requirements are zero. On the other hand, the well-known BIN-PACKING problem is a special case of the resource constrained scheduling problem. The hybrid nature of the problem is also reflected in our algorithms. To tackle it, we combine ideas and techniques from both domains.

In this work we study several variations of the problem. These include different machine models (i.e., all machines are identical, or processing times and resource requirements of jobs change for different machines), and also the types of schedules we allow (preemption, migration etc.). As for the avionics optimization problems, we analyze the algorithmic complexity of these variations by giving a rigorous account on the approximation landscape. Along the way we improve on a classical result by Garey and Graham [GG75] from 1975. We study two fundamentally different types of algorithms: *Online* and *offline* algorithms. Online algorithms are less powerful in the sense that they learn about a problem instance over time, allowing only limited adaptations to a previously computed schedule once additional information about an instance is revealed to the algorithm. In contrast, offline algorithms have complete information about the problem instance. We also discuss how approximation algorithms for online algorithms can be translated into approximation algorithms for certain types of real-time scheduling problems, which completes the circle: the thesis both starts and ends with the study of real-time scheduling problems. For details on our results regarding the resource constrained scheduling problem, we refer to Section 4.1.3 and Section 5.1.3.

1.3 Organization

This document consists of two main parts. Chapters 2 and 3 deal with the real-time avionics optimization problem. Chapter 2 focuses on our theoretical results in that area. In Chapter 3 we then discuss how to exploit our theoretical structural insights to derive algorithms that quickly compute optimal solutions for real-world instances. The second part of the thesis consists of Chapter 4 and Chapter 5. Here we study the resource constrained scheduling problems. First in Chapter 4 we present our results regarding offline scheduling, whereas Chapter 5 is devoted to online scheduling and real-time scheduling. The two parts of the thesis are organized so that they are self-contained, in particular they do not build on each other and can be read in arbitrary order. Also the individual chapters are kept to be rather self-contained. The reader is advised however to read the respective introductory sections of the two parts, i.e., Section 2.1 for part one and Section 4.1, before proceeding to the second chapters of the respective parts. We recommend however to read Section 1.5 first for a brief introduction on fundamental concepts and notation regarding approximation algorithms.

1.4 Sources

The results presented in Chapter 2 are joint work with Eisenbrand, Hähnle, Skutella, Verschae and Wiese. The chapter is based on the paper [EHN+10]. The practical results from Chapter 3 are based on the paper [EKM+10] and are joint work with Eisenbrand, Kesavan, Mattikalli, Nordsieck, Skutella, Verschae and Wiese. The results presented in Chapter 4 have several sources. The work in Section 4.2 of that chapter regarding identical machines is joint work with Andreas Wiese, and the presentation is based on the paper [NW12]. The results regarding unrelated machines that we discuss in Section 4.3 is joint work both with Sanjoy Baruah and Andreas Wiese and based on the paper [BNW11]. Finally the results presented in Chapter 5 are joint work with José Verschae and Andreas Wiese. These results are not yet published at the time of writing the thesis. At various points in the thesis we also review related work by other authors. The respective sections are clearly marked as such, and credit to the authors is given.

1.5 Prerequisites

We assume that the reader is familiar with basic concepts from theoretical computer science, in particular from complexity theory. These include the complexity classes P and NP and concepts like polynomial-time reduction, NP-hardness and NP-completeness. Also the $O(\cdot)$, $o(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$ notation is used throughout the thesis. Two good textbooks that introduce these concepts are a book by Cormen, Leiserson, Rivest and Stein [CLRS01] and a book by Papadimitriou [Pap94]. We also assume that the reader is accustomed to the basics of linear programming. I.e., the reader should know what a linear program (LP) is and that it can be solved efficiently. In depth knowledge of these topics is not required, though. We refer to a book by Bertsimas and Tsitsiklis [BT97] for an extensive introduction.

We now give a brief introduction into basic terminology used in conjunction with approximation algorithms. People familiar with this subject can safely skip the remainder of this section. We say that an algorithm *runs in polynomial time* if its running time can be upper bounded by a polynomial f of constant degree: For every input instance I, if $|I|$ denotes the (binary) encoding length of the instance, the running time of the algorithm is at most $f(|I|)$. We equivalently say that algorithms that run in polynomial time are *efficient*. Given some minimization problem, for every instance I of the problem, with $OPT(I)$ we denote the minimum value of a feasible solution to the problem. An algorithm is a c-approximation algorithm, if it is efficient and, for all instances I, computes a solution of value at most $c \cdot OPT(I)$. As an example, consider a polynomial time algorithm that computes feasible schedules for some variant of the resource constrained scheduling problem. Assume that it has the property that for every instance it computes a schedule that is at most twice as long as an optimal schedule. Then that algorithm is a 2-approximation algorithm. An *asymptotic* c-approximation algorithm is an algorithm that it is efficient and, for all instances I, computes a solution of value at most $c \cdot OPT(I) + o(OPT(I))$. In other words, for an asymptotic approximation algorithm we allow an additive error that, for instances whose optimal solution is large enough, gets negligible compared to the value of the optimal solution. For our example, an algorithm that, for all instances, computes a schedule of makespan at most $2 \cdot OPT + 10000 \cdot \sqrt{OPT}$, where OPT denotes the makespan of an optimal schedule, would be an asymptotic 2-approximation algorithm. A *c-polynomial time approximation scheme* (c-PTAS) is a family of approximation algorithms of the following form. Given a minimization problem, for every constant $\epsilon > 0$ there is an approximation algorithm in the family that has an approximation factor of $c + \epsilon$. Note that as ϵ is constant, the running time of the algorithm can depend on ϵ in an arbitrary way. However its running time has to be polynomial in the input size. For simplicity of notation, a 1-PTAS is simply referred to as a PTAS. A *c-fully polynomial time approximation scheme* (c-FPTAS) is a c-PTAS, where the running time of each $(c + \epsilon)$-approximation algorithm is also polynomial in the approximation guarantee $\frac{1}{\epsilon}$. A 1-FPTAS is simply referred to as an FPTAS. Note that for practical applications, an FPTAS is more desirable over a PTAS: if the dependence on the approximation quality is exponential (which is allowed for PTASes), the algorithm is usually not very well suited for any practical purposes. A problem is *APX-hard* if there is a constant $c > 1$ so that there is no c-approximation algorithm for the problem unless $P = NP$. We also say that such a problem is *inapproximable within a factor of c*. Observe that APX-hardness of a problem implies that there is no PTAS (with approximation factor one).

A problem is *NP-hard in the strong sense* (or *strongly NP-hard* for short), if it is NP-hard even if the input is given in unary encoding, i.e., a number $\ell \in \mathbb{Z}$ in the input is encoded using ℓ bits instead of $\log(\ell)$ bits. For example, the BIN-PACKING problem is strongly NP-hard [GJ79], whereas PARTITION is not: for the latter, there is an algorithm based on dynamic programming that solves the problem in time polynomial in the unary encoding length of the instance [GJ79]. An algorithm whose running time is bounded in this way is said to run in *pseudopolynomial time*. It is also worth mentioning that strongly NP-hard problems do not admit an FPTAS, unless $P = NP$ [GJ78]. For an extensive introduction into the concepts of approximation algorithms, we refer to a book by Vazirani [Vaz01].

2 Real-Time Avionics Optimization

2.1 Introduction

We now study a problem that occurs in the avionics industry in the design of modern aircraft, as introduced in Section 1.1. Recall the definition of the model: one is given a set of *tasks* $\mathcal{T}$, where each task $\tau \in \mathcal{T}$ is characterized by its *execution time* $c(\tau) \in \mathbb{N}$ and *period* $p(\tau) \in \mathbb{N}$. The goal is to assign the tasks to identical machines and to compute offsets $a(\tau) \in \mathbb{N}_0$ such that no collision occurs. A task τ generates one *job* with execution time $c(\tau)$ at every time unit $a(\tau) + p(\tau) \cdot k$ for all $k \in \mathbb{N}_0$. Each job needs to be processed immediately and non-preemptively after its generation on the task's machine. A collision occurs if two jobs are simultaneously active on one machine. In the *periodic maintenance problem*, given a task set $\mathcal{T}$, the goal is to compute feasible offsets $a(\tau)$ for all tasks of the instance, or to assert that no such offsets exist. Figure 1.1 from the introduction shows an example with three tasks τ_1, τ_2, τ_3. The execution times are $c(\tau_1) = c(\tau_2) = 1$ and $c(\tau_3) = 2$. The periods are $p(\tau_1) = 6$, $p(\tau_2) = 10$ and $p(\tau_3) = 15$. The upper part of the picture shows an infeasible assignment of offsets ($a(\tau_1) = 0$, $a(\tau_2) = 1$, $a(\tau_3) = 2$) whereas the lower part shows a feasible assignment of offsets ($a(\tau_1) = 1$, $a(\tau_2) = 0$, $a(\tau_3) = 2$). The problem occurring in real-world systems is more complex as usually the instances are too large to be scheduled to a single machine. For that reason, the engineers are interested in the corresponding *machine minimization* problem, i.e., they want to find the minimum number of identical machines on which the tasks can be distributed in a feasible way. We refer to this problem as the *periodic maintenance minimization problem*.

This chapter is devoted to the complexity analysis of the periodic maintenance (minimization) problem. We characterize the algorithmic complexity by presenting approximation algorithms and establishing inapproximability results. Insights about the problem structure gained in this analysis also lay the foundations for our results regarding practical applications presented in Chapter 3.

2.1.1 Related Work

To the best knowledge of the author, there are two independent branches of research regarding the periodic maintenance problem. The problem first appeared in a paper by Wei and Liu [WL83] from 1983. They studied the special case of instances with unit execution times and give a sufficient (but not necessary) condition on the existence of a schedule for instances with unit execution times. In a follow up paper from 1990, Baruah et al. [BRTV90] show that the problem is NP-hard (in the strong sense, also with unit execution times). Independently it is shown in [BBNS02, Bha98] that the problem is NP-hard (in the strong sense).

In an unrelated line of research, Korst et al. [KALW91, KAL96] study the periodic maintenance minimization problem under the name of “constrained scheduling problem”. In contrast to the other branch, they consider arbitrary execution times and a multi-machine variant similar to ours. Their focus is on designing heuristics for the problem. They also prove that the problem is NP-hard.

The periodic maintenance minimization problem is a generalization of the BIN-PACKING PROBLEM (e.g., see [Hoc96, Vaz01]). In the BIN-PACKING problem, one is given a set of items, each item i having an item size $s(i)$ with $0 \le s(i) \le 1$. The goal is to find a partition of the items into a minimum number of bins, where in each bin, the sum of item sizes must not exceed 1. Observe that given a periodic maintenance instance where the period lengths of all tasks are identical, the problem is equivalent to BIN-PACKING. However, the additional generality of the periodic maintenance problem leads to an increased computational complexity of the problem. In particular, it is possible to approximate BIN-PACKING with an approximation ratio of 1.5 [SL94] whereas, as we show later, it is impossible to approximate periodic maintenance minimization with almost any nontrivial factor, unless $P = NP$.

2.1.2 Our Contribution

We give a rigorous account on the complexity landscape of the periodic maintenance minimization problem, both designing approximation algorithms and proving inapproximability results. It turns out that the periodic maintenance minimization problem is intractable. We show that the problem is NP-hard to approximate within a factor of $n^{1-\epsilon}$ for any constant $\epsilon > 0$, where n denotes the number of tasks in the instance. This is a very negative result in terms of approximation algorithms as it essentially shows that one cannot do better than trivial approximation algorithms: a trivial n-approximation algorithm is to open a separate machine for each task of the instance. Even worse, it is not even straightforward to design meaningful heuristics. A typical First-Fit heuristic would assign tasks one at a time, trying to *add* tasks to machines already opened, then opening a new machine if a task cannot be added to any open machine. But already the problem of deciding whether a task can be added to a machine is difficult: Given a set of tasks with feasible offsets on a single machine, and another task τ. The problem of deciding whether there is a feasible offset $a(\tau)$ for τ on that machine is NP-complete. Fortunately, real-world instances usually have a strong property: the task

Periods	Approximation guarantee	Hardness results	
		absolute	asymptotic
arbitrary	n	$n^{1-\epsilon} OPT$	———
harmonic	2	$(2-\epsilon)\, OPT$	$(2-\epsilon)\, OPT + o(OPT)$

Table 2.1: Overview of our results regarding the periodic maintenance minimization problem.

periods are *harmonic*, i.e., for any two periods $p(\tau_1)$ and $p(\tau_2)$, $\tau_1, \tau_2 \in \mathcal{T}$, one period is an integer multiple of the other. We develop a special data structure, the (compressed) bin trees. For harmonic instances, with the help of bin trees, one can implement efficient First-Fit type heuristics. We present such a heuristic and show that it is a 2-approximation algorithm. We also give a matching inapproximability result by showing that it is NP-hard to approximate within a factor of $(2-\epsilon)$, for every constant $\epsilon > 0$. Unlike the BIN-PACKING problem where there is an asymptotic PTAS, this inapproximability bound also holds asymptotically. See Table 2.1 for an overview of our results.

We also consider a variation of the problem, which we call the *resource augmentation variant*. Here, the number m of machines is given as part of the input instance $\mathcal{T}$. We assume that a feasible solution that uses at most m machines exists. Instead of opening additional machines to find a feasible solution, we allow to increase the *machine speed*: For $\alpha \geq 1$, a task $\tau \in \mathcal{T}$ only needs execution time of $p(\tau)/\alpha$. An α-approximation algorithm for this variant is a polynomial time algorithm that always finds a feasible solution on m speed-α machines, provided that there is a feasible solution on speed 1 machines. It turns out that with slight adaptation of the methods, we essentially get the same results in terms of approximation ratios as in the machine minimization variant: The general problem is NP-hard to approximate within a factor of $n^{1-\epsilon}$ for any constant $\epsilon > 0$. For the harmonic sub-case, there is a 2-approximation algorithm. Here we do not have a matching inapproximability result for the harmonic case though.

2.1.3 Outline

This chapter is organized as follows. First in Section 2.2 we deal with the general (i.e., non-harmonic) case. We introduce general notion and concepts and establish the $n^{1-\epsilon}$ inapproximability and related hardness results. Afterwards in Section 2.3, we discuss the harmonic case. We introduce the data structure of *bin-trees*, present the 2-approximation algorithm and the hardness results. Finally in Section 2.4 we give an outlook on possible future research directions. Unless stated otherwise, we deal with the periodic maintenance minimization problem, i.e., the machine minimization variant. At some points we discuss adaptations to make our methods work also for the resource augmentation variant.

2.2 The General Case

Properties of schedules for the periodic maintenance problem can be well described using fundamental notion and methods from modular arithmetic. For that reason, we start with a brief excursion before discussing the periodic maintenance problem.

2.2.1 Prerequisites from Modular Arithmetic

Given numbers $n \in Z$, $m \in \mathbb{Z} \setminus \{0\}$ we say that m *divides* n, or $m|n$ in short, if n is an integer multiple of m. This means that there is a number $k \in \mathbb{Z}$ so that $n = k \cdot m$. The number $n \bmod m$ is defined as the *unique* number $k \in \mathbb{Z}$ with $0 \leq k < m$ so that $m|(n-k)$. In other words, $n \bmod m$ is the *remainder* of the (integer) division of n by m. The operation mod defines an equivalence relation $\equiv$ over the integers, the *congruence* relation: For $n, n' \in \mathbb{Z}$ and $m \in \mathbb{Z} \setminus \{0\}$ we say that $n \equiv n' \pmod m$ if $n \bmod m = n' \bmod m$, or equivalently if $m|(n-n')$. Given numbers $k_1, \ldots, k_n \in \mathbb{Z}$, the *greatest common divisor* $g := \gcd\{k_1, \ldots, k_n\} \in \mathbb{N}_0$ is the natural number satisfying the following properties:

- $g|k_i$ for all $i = 1, \ldots, n$.
- If $p \in \mathbb{Z} \setminus \{0\}$ so that $p|k_i$ for all $i = 1, \ldots, n$, then $p|g$.

Similarly the *least common multiple* $\ell := \operatorname{lcm}\{k_1, \ldots, k_n\} \in \mathbb{N}_0$ is the natural number satisfying

- $k_i|\ell$ for all $i = 1, \ldots, n$
- If $m \in \mathbb{Z} \setminus \{0\}$ so that $k_i|m$ for all $i = 1 \ldots, n$, then $\ell|m$.

A number $p \in \mathbb{N}$, $p > 1$, is *prime* if there is no $m \in \{2, 3, \ldots, p-1\}$ so that $m|p$. Two numbers $p, q \in \mathbb{N}$ are co-prime, if $\gcd\{p, q\} = 1$. A fundamental theorem from number theory is the following.

Theorem 2.2.1 (Chinese remainder theorem (simplified))**.** *Given a set of numbers $p_1, \ldots, p_n \in \mathbb{N}$ and numbers $a_1, \ldots, a_n \in \mathbb{Z}$. Assume that $p_1, \ldots, p_n$ are pairwise co-prime. There is a number $a \in \mathbb{Z}$ that simultaneously satisfies the following congruences:*

$$a \equiv a_i \pmod{p_i} \ \forall\ i = 1, \ldots, n.$$

Given a number $k \in \mathbb{Z}$, the *ideal* generated by k is the set $k \cdot \mathbb{Z} := \{k \cdot m \ : \ m \in \mathbb{Z}\}$. Note that $\ell \in k \cdot \mathbb{Z}$ if and only if $\ell \equiv 0 \bmod k$. For two sets of numbers I_1 and I_2 we define $I_1 + I_2 := \{i_1 + i_2 \ : \ i_1 \in I_1,\ i_2 \in I_2\}$. We conclude the brief excursion stating an elementary number theoretical observation that follows from Bézout's identity.

Lemma 2.2.2. *Given $k_1, k_2 \in \mathbb{Z}$, then $k_1 \cdot \mathbb{Z} + k_2 \cdot \mathbb{Z} = \gcd(k_1, k_2) \cdot \mathbb{Z}$*

2.2.2 General Concepts

We now come back to the periodic maintenance problem and review some basic facts. These basic observations also appear in one form or another in previous work [WL83, BRTV90, BBNS02, Bha98, KALW91, KAL96]. Consider an instance $\mathcal{T}$. Note that every schedule is periodic: it repeats after $\text{lcm}\{p(\tau) : \tau \in \mathcal{T}\}$ many time units. This immediately leads to a feasibility test, which however is not efficient. A polynomial time feasibility test can be derived as follows. Let $\tau_1, \tau_2 \in \mathcal{T}$ be two tasks which we want to assign to a single machine. Assume that $c(\tau_1) = c(\tau_2) = 1$. How can we check whether two offsets $a(\tau_1)$ and $a(\tau_2)$ lead to a collision? Because the execution times are one and periods are integer, the two tasks collide if and only if they release a job simultaneously. Hence they collide if and only if there are numbers $\ell_1, \ell_2 \in \mathbb{N}_0$ so that

$$a(\tau_1) + \ell_1 \cdot p(\tau_1) = a(\tau_2) + \ell_2 \cdot p(\tau_2). \tag{2.1}$$

Note a subtlety here. We can relax the range of ℓ_1, ℓ_2 to be from the integers instead of $\mathbb{N}_0$. This is because of the following. If there are $\ell_1, \ell_2 \in \mathbb{Z}$ that satisfy Equation (2.1), then also $\ell_1' := \ell_1 + k \cdot \frac{\text{lcm}(p(\tau_1),\, p(\tau_2))}{p(\tau_1)} \in \mathbb{Z}$ and $\ell_2' := \ell_2 + k \cdot \frac{\text{lcm}(p(\tau_1),\, p(\tau_2))}{p(\tau_2)} \in \mathbb{Z}$ satisfy the equation for any $k \in \mathbb{Z}$. Hence the existence of $\ell_1, \ell_2 \in \mathbb{Z}$ satisfying Equation (2.1) implies the existence of natural numbers satisfying the equation. Coming back to our feasibility test, we conclude that the tasks τ_1 and τ_2 collide if and only if there are $\ell_1, \ell_2 \in \mathbb{Z}$ satisfying Equation (2.1). Reformulating this we get that they collide if and only if

$$a(\tau_2) - a(\tau_1) = \ell_1 \cdot p(\tau_1) - \ell_2 \cdot p(\tau_2).$$

We conclude that they collide if and only if

$$a(\tau_2) - a(\tau_1) \in p(\tau_1) \cdot \mathbb{Z} + p(\tau_2) \cdot \mathbb{Z} = \gcd(p(\tau_1), p(\tau_2)) \cdot \mathbb{Z},$$

using Lemma 2.2.2 for the last equality. Hence $\gcd(p(\tau_1), p(\tau_2)) | (a(\tau_2) - a(\tau_1))$. We get the following feasibility criterion.

Lemma 2.2.3. *Two tasks τ_1, τ_2 with $c(\tau_1) = c(\tau_2) = 1$ collide if and only if*

$$a(\tau_1) \equiv a(\tau_2) (\bmod \gcd(p(\tau_1), p(\tau_2))).$$

As also observed by Korst et al. [KALW91], this can be generalized easily to the case of arbitrary execution times as follows.

Lemma 2.2.4 ([KALW91])**.** *Two tasks τ_1, τ_2* do not *collide if and only if*

$$c(\tau_1) \leq (a(\tau_2) - a(\tau_1)) \bmod \gcd(p(\tau_1), p(\tau_2)) \leq \gcd(p(\tau_1), p(\tau_2)) - c(\tau_2)$$

Proof. Replace the tasks τ_1 (τ_2) with $c(\tau_1)$ ($c(\tau_2)$) many tasks of execution time 1 and set the offsets so that they are executed one after another: set $c(\tau_1^i) = 1$, $p(\tau_1^i) = p(\tau_1)$ and $a(\tau_1^i) =$

$a(\tau_1) + i$ for all $i = 0, \dots, c(\tau_1) - 1$. Analogously set $c(\tau_2^i) = 1$, $p(\tau_2^i) = p(\tau_2)$ and $a(\tau_2^i) = a(\tau_2) + i$ for all $i = 0, \dots, c(\tau_2) - 1$. Observe that τ_1 and τ_2 collide if and only if one of the pairs τ_1^i, τ_2^j collides. By Lemma 2.2.3 they collide if and only if $0 \equiv a(\tau_2^j) - a(\tau_1^i) = a(\tau_2) - a(\tau_1) + j - i (\operatorname{mod} \gcd(p(\tau_1), p(\tau_2)))$, i.e.,

$$a(\tau_2) - a(\tau_1) \bmod \gcd(p(\tau_1), p(\tau_2)) = i - j \bmod \gcd(p(\tau_1), p(\tau_2)).$$

Tasks τ_1 and τ_2 do *not* collide if none of these equations is satisfied. Observe that $i - j$ ranges from $-c(\tau_2) + 1$ to $c(\tau_1) - 1$. Hence in order to be collision free, the number $a(\tau_2) - a(\tau_1) \bmod \gcd(p(\tau_1), p(\tau_2))$ must be lower bounded by $c(\tau_1)$ and upper bounded by the number $-c(\tau_2) \bmod \gcd(p(\tau_1), p(\tau_2)) = \gcd(p(\tau_1), p(\tau_2)) - c(\tau_2)$. □

Analogous to the basic problem of BIN-PACKING, we can derive a lower bound on the optimal solution as follows. In BIN-PACKING, the sum of item sizes rounded up to the nearest integer gives a lower bound on the number of bins needed in any solution, simply by the fact that every bin can hold items of cumulative size at most 1. An analogous statement can be derived for the periodic maintenance problem. To this end, we define the *utilization* $\operatorname{util}(\tau)$ of a task $\tau \in \mathcal{T}$ as the execution time per period ratio:

$$\operatorname{util}(\tau) := \frac{c(\tau)}{p(\tau)}.$$

For an instance $\mathcal{T}$, we define $\operatorname{util}(\mathcal{T}) := \sum_{\tau \in \mathcal{T}} \operatorname{util}(\tau)$. Let $OPT(\mathcal{T})$ denote the number of machines used in an optimal solution. We get the following lower bound.

Lemma 2.2.5. *Given an instance $\mathcal{T}$, there is no feasible one-machine schedule if* $\operatorname{util}(\mathcal{T}) > 1$. *Moreover, $OPT(\mathcal{T}) \geq \lceil \operatorname{util}(\mathcal{T}) \rceil$.*

Proof. Let $k := \operatorname{lcm}\{p(\tau) : \tau \in \mathcal{T}\}$ be the least common multiple of all periods of the instance. Then each task τ_j runs jobs of total execution time exactly $c(\tau) \cdot \frac{k}{p(\tau)} = k \cdot \operatorname{util}(\tau)$ during the time frame $[0, k)$. Thus no matter how the offsets are assigned, the total execution time needed for the instance $\mathcal{T}$ during that time-frame is $k \cdot \operatorname{util}(\tau)$. Thus if $\operatorname{util}(\mathcal{T}) > 1$, there can be no single machine schedule.

The second claim easily follows from the first. In order to obtain a feasible schedule, every solution has to partition the tasks to machines so that in no group, the total utilization exceeds 1. We conclude that the optimal solution needs at least $\lceil \operatorname{util}(\mathcal{T}) \rceil$ many machines. □

Note that the utilization bound can be arbitrarily bad. Take any instance with unit execution times, where the periods are sufficiently large pairwise disjoint primes. Then the utilization bound is 1. The feasibility criterion from Lemma 2.2.3 however implies that each task needs a separate machine, i.e., $OPT(\mathcal{T}) = n$.

2.2.3 Popular heuristics fail for periodic maintenance

Consider an instance $\mathcal{T}$. *First-Fit* is a simple and very popular heuristic for many packing problems, such as, e.g., BIN-PACKING etc. Adapted to our problem, First-Fit considers tasks in some given order and greedily packs the current task on the first open machine on which it fits. In case there is no such machine, it opens a new machine on which the current task is scheduled. A crucial subproblem occurring in this context is to decide whether a task τ can be scheduled on a machine on which other tasks, say $\tau_1, \ldots, \tau_n$, have already been scheduled. Determining whether there is a feasible one-machine schedule for $\tau_1, \ldots, \tau_n, \tau$ is an NP-hard problem. But what about a simpler variant? Assume that we already have feasible offsets for tasks $\tau_1, \ldots, \tau_n$. We want to assign an offset a for τ without changing the offsets of the other tasks. If all processing times are equal to 1 and the period of τ is p, the feasible offsets a for task τ are the solutions to the system

$$a \not\equiv a_j \mod \gcd(p, p(\tau_j)) \quad \text{for } j = 1, \ldots, n.$$

For arbitrary $p_j \in \mathbb{N}$ and $p = \prod_{i=1}^{n} p(\tau_j)$, this is the NP-complete problem of computing *simultaneous incongruences*, see, e.g., [GJ79]. Thus, already this crucial subproblem is NP-hard.

One heuristical way around this is as follows: Split the instance into several subproblems, then treat each of the subproblems independently. Here we can group the tasks of an instance by their period length and schedule each group to different machines. This way, for each group, we "only" need to solve a BIN-PACKING problem as all periods of each subproblem are identical. Unfortunately this heuristic is only an n-approximation algorithm: for any instance that has a feasible one-machine schedule but n different periods, our algorithm would open n machines. Of course such an approximation guarantee is far from desirable. Unfortunately one cannot do (much) better as we show next.

2.2.4 Inapproximability

We now prove that the periodic maintenance minimization problem is NP-hard to approximate within a factor of $n^{1-\epsilon}$ for any constant $\epsilon > 0$. This is established by a reduction from a classical problem from graph theory. A *graph* $G = (V, E)$ is a set of *nodes* V together with a set of *edges* $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$. An independent set in G is a set of vertices $S \subseteq V$ such that no two nodes from S are connected by an edge.

Definition 2.2.6 (COLORING problem). *Given a graph $G = (V, E)$, find a partition of V into a minimum number of independent sets.*

A result by Zuckerman [Zuc07] shows that the COLORING problem is NP-hard to approximate within a factor of $n^{1-\varepsilon}$ for any constant $\epsilon > 0$. Here n is the number of nodes of the graph. We will give a reduction from COLORING to the periodic maintenance minimization problem that preserves approximation ratios and hence allows to transfer the inapproximability results for

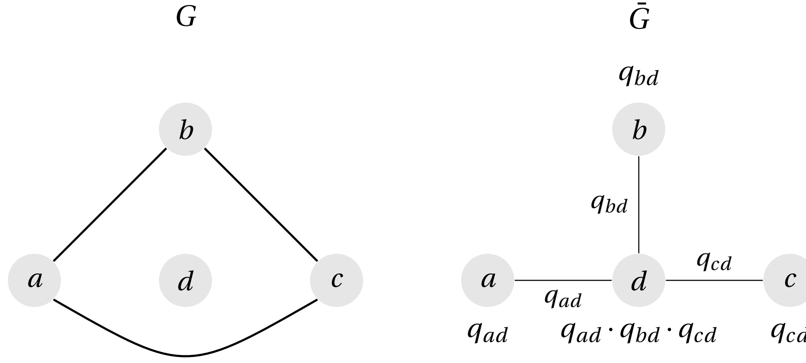

Figure 2.1: Reduction of the COLORING problem to the machine minimization problem.

coloring to the machine minimization problem.

The reduction from COLORING

We remark that this reduction has been stated independently in the PhD-thesis of Bhatia [Bha98]. It works as follows. Let $G = (V, E)$ be an instance for the coloring problem. Let $\bar{E}$ be the complement of E, i.e.,

$$\bar{E} := \Big\{ \{u, v\} \,:\, u, v \in V, \{u, v\} \notin E \Big\}.$$

We construct a periodic maintenance instance as follows. We choose $|\bar{E}|$ many different primes, a prime q_e for each $e \in \bar{E}$. For each node $v \in V$, we define a task τ_v with $c(\tau_v) := 1$ and set its period to the product of primes from adjacent edges in $\bar{E}$, i.e.,

$$p(\tau_v) := \prod_{e \in \bar{E}:\ v \in e} q_e.$$

We denote with $\text{red}(G) := \{\tau_v \,:\, v \in V\}$ the periodic maintenance instance obtained from graph G using this construction. See Figure 2.1 for an illustration of the reduction. Note that we can compute $\text{red}(G)$ in polynomial time. In particular, we can find the primes q_e in polynomial time, since the Prime Number Theorem guarantees $\Theta\left(\frac{x}{\ln(x)}\right)$ primes among the first x natural numbers (see e.g., [New80]).

The construction has two important properties:

Lemma 2.2.7. *Given a graph $G = (V, E)$, the machine minimization instance* $\text{red}(G)$ *satisfies:*

a) For each edge $\{u, v\} \in E$, the tasks τ_u and τ_v cannot be assigned to the same machine.

b) If $U \subseteq V$ is an independent set, then $\{\tau_v \,:\, v \in U\}$ can be scheduled to one machine.

Proof. For each node $v \in V$, define $E_v := \{e \in \bar{E} :\ v \in e\}$. Observe that by construction, for each

pair u, v of nodes we have

$$\gcd(p(\tau_u), p(\tau_v)) = \prod_{e \in E_u \cap E_v} q_e \tag{2.2}$$

a) Let $u, v \in V$ such that $\{u, v\} \in E$. Thus $\{u, v\} \notin \bar{E}$, and hence $E_u \cap E_v = \emptyset$. Thus (2.2) implies $\gcd(p(\tau_u), p(\tau_v)) = 1$. Assume that u and v can be scheduled to the same machine. Using the feasibility criterion from Lemma 2.2.3, then there are offsets $a(\tau_u), a(\tau_v) \in \mathbb{Z}$ such that

$$\mathbb{Z} \ni a(\tau_v) - a(\tau_u) \notin \gcd(p_u, p_v)\mathbb{Z} = \mathbb{Z},$$

a contradiction.

b) Let U be an independent set in G. We can define valid offsets as follows. For each edge $e = \{u, v\} \in \bar{E}$, set $\delta_{e,u} = 0$ and $\delta_{e,v} = 1$ (choose the roles of u and v arbitrarily). Now for each $v \in U$, let the offsets $a(\tau_v)$ be the solution of the following system of simultaneous congruences:

$$a(\tau_v) \equiv \delta_{e,v} \bmod q_e \quad \forall e \in \bar{E},\ v \in e.$$

The Chinese Remainder Theorem [JJ98] guarantees the existence of these $a(\tau_v)$. Let $u, v \in U$, $u \neq v$. Because U is an independent set, we have $e = \{u, v\} \in \bar{E}$. By definition of the offsets, we have $a(\tau_u) \equiv 0 \bmod q_e$ and $a(\tau_v) \equiv 1 \bmod q_e$ (or vice versa). In particular

$$a(\tau_u) \not\equiv a(\tau_v) \bmod q_e.$$

As $E_u \cap E_v = \{e\}$ we conclude $\gcd(p(\tau_u), p(\tau_v)) = q_e$. Hence Lemma 2.2.3 asserts that no collision occurs.

□

We can now prove that our construction is a reduction:

Lemma 2.2.8. *For each $k \in \mathbb{N}$, a graph G is k-colorable if and only if* red(G) *can be scheduled to k machines.*

Proof. Assume that red(G) can be scheduled to k machines. For each machine $i = 1, \dots, k$, let

$$V_i := \{v \in V \ :\ \tau_v \text{ assigned to machine } i\}$$

be the set of nodes whose tasks are assigned to the machine. Clearly $V_1, \dots, V_k$ is a partition of V. Moreover with Lemma 2.2.7 we know that no two nodes $u, v \in V_i$ are adjacent, for every $i = 1, \dots, k$. Thus each of the sets V_i is independent and they form a k-coloring. Now assume that G is k colorable, and let $V_1, \dots, V_k$ be a partition of V into independent sets. With Lemma

2.2.7 we know that for each V_i, we can assign all tasks emerging from its nodes to a single machine. Hence we need no more than k machines for a feasible schedule. □

We stress again that this reduction only uses tasks of unit execution time.

For the variant of resource augmentation, a very similar reduction works. Recall that in this variant, instead of opening new machines, we are allowed to increase the speed of machines. Given a graph $G = (V, E)$ and a number $k \in \mathbb{N}$. Let $\bar{E}$ denote the complement of E as defined above. For each $e \in \bar{E}$, we choose pairwise different primes $q_e > k$. For each node $v \in V$, we define a task τ_v with $c(\tau_v) := 1$ and

$$p(\tau_v) := k \cdot \prod_{e \in \bar{E}:\ v \in e} q_e.$$

We denote with $\widetilde{\mathrm{red}}(G) := \{\tau_v \ : \ v \in V\}$ the periodic maintenance instance obtained from graph G using this construction. Note that it is essentially the same construction as in the machine minimization case, except that the all periods are multiplied with a factor of k. We can show that it is a reduction.

Lemma 2.2.9. *If a graph $G = (V, E)$ is k-colorable, then there is a feasible single machine schedule for $\widetilde{\mathrm{red}}(G)$ (on a speed-1 machine).*

Proof. For each $v \in V$, let $\chi(v) \in \{0, \ldots, k-1\}$ be its color. For each edge $e = \{u, v\} \in \bar{E}$, set $\delta_{e,u} = 0$ and $\delta_{e,v} = 1$ (choose the roles of u and v arbitrarily). Now for each $v \in U$, let the offsets $a(\tau_v)$ be the solution of the following system of simultaneous congruences:

$$\begin{aligned} a(\tau_v) &\equiv \chi(v) \bmod k \\ a(\tau_v) &\equiv \delta_{e,v} \bmod q_e \quad \forall e \in \bar{E},\ v \in e. \end{aligned}$$

The Chinese Remainder Theorem [JJ98] guarantees the existence of these $a(\tau_v)$ because all modules q_e and k are pairwise co-prime. Let $u, v \in V$, $u \neq v$. If $e = \{u, v\} \in E$, then as χ is a coloring we have $\chi(u) \neq \chi(v)$. Hence by definition of the offsets we get $a(\tau_u) - a(\tau_v) \equiv \chi(u) - \chi(v) \not\equiv 0 (\bmod k)$. As by construction we have $\gcd(p(\tau_u), p(\tau_v) = k$, we conclude that tasks τ_u and τ_v do not collide.

If however $e = \{u, v\} \notin E$, assume that τ_u and τ_v do collide. Hence we have that $a(\tau_v) \equiv a(\tau_u) \bmod \gcd(p(\tau_u), p(\tau_v))$. Here $\gcd(p(\tau_u), p(\tau_v)) = k \cdot q_e$ and thus in particular $a(\tau_v) \equiv a(\tau_u) \bmod q_e$, in contradiction to the construction of the a. We conclude that no collision occurs and hence the schedule is feasible. □

Lemma 2.2.10. *If there is a feasible single machine schedule for $\widetilde{\mathrm{red}}(G)$ (on a speed-1 machine), then the graph $G = (V, E)$ is k-colorable.*

Proof. For each node $v \in V$, set $\chi(v) := a(\tau_v) \bmod k$. Now consider two nodes $u, v \in v$, $u \neq v$ with $\{u, v\} \in E$. We need to show that $\chi(u) \neq \chi(v)$. By construction, $\gcd(p(\tau_u), p(\tau_v) = k$. As the tasks do not collide, by Lemma 2.2.3 we have that $\chi(u) \not\equiv \chi(v) \bmod k$, which shows the claim. □

Implications for approximation hardness

The reduction creates exactly one task for each node of the graph. Thus Theorem 2.2.8 shows that inapproximability results in terms of numbers of nodes can be transferred directly to inapproximability results for the machine minimization problems in terms of number of tasks. We transfer the following result from Zuckerman [Zuc07], which states that the chromatic number, i.e., the minimal number of colors needed to color a graph, is hard to approximate:

Theorem 2.2.11 ([Zuc07])**.** *It is NP-hard to approximate the chromatic number within a factor of $n^{1-\varepsilon}$ for every constant $\varepsilon > 0$, where n is the number of nodes in the graph.*

With the discussion above, the following is immediate:

Theorem 2.2.12. *The machine minimization problem (even for unit execution times) is hard to approximate within a factor of $n^{1-\varepsilon}$ for every constant $\varepsilon > 0$, unless $P = NP$.*

For the resource augmentation variant, Lemma 2.2.9 and Lemma 2.2.10 imply an analogous result.

Theorem 2.2.13. *The resource augmentation variant (even for unit execution times) is hard to approximate within a factor of $n^{1-\varepsilon}$ for every constant $\varepsilon > 0$, unless $P = NP$.*

Proof. Assume that we have an α-approximation algorithm for the resource augmentation problem. We can turn it into an α-approximation algorithm for graph coloring as follows. For each $k = 1, \dots, n$, we construct the instance $\widetilde{\text{red}}(G)$ for the resource augmentation problem. If G is k-colorable, the two lemmas from above guarantee a that a feasible single machine schedule exists, and hence the approximation algorithm can compute one on a speed-α machine. We run the algorithm for every k to find the smallest one where the algorithm succeeds. Note that the fact that the algorithm finds a speed-α schedule does *not* imply that there is a feasible speed-one schedule. However, it is not hard to see that instead it implies that there is a feasible speed-one machine schedule for the instance $\widetilde{\text{red}}(G)$ where we choose $\tilde{k} := \lfloor k \cdot \alpha \rfloor$. We skip the formal proof for this claim as it is a bit technical. The essential idea is that the feasible offsets from the speed-α schedule give feasible offsets for a speed-one schedule of the new instance by scaling the offsets up by a factor of α and rounding it down to the nearest integer.

This in turn shows (with Lemma 2.2.10) that the graph is $\lfloor \alpha \cdot k \rfloor$ colorable. Hence we can α-approximate the coloring number of the graph. We conclude that there can be no $n^{1-\varepsilon}$-approximation algorithm for the resource augmentation problem. □

Figure 2.2: A schedule for a single machine. The gray jobs belong to the task with period length q_1, the striped jobs to tasks with period length $q_2 = 3 \cdot q_1$, and the dotted jobs to tasks with period length $q_3 = 6 \cdot q_1$.

2.3 The Harmonic Case

As pointed out in Section 2.2.3, it is not clear how to design an efficient First-Fit type algorithm for the (general) periodic maintenance problem as already the subproblem of deciding whether a task can be added to a machine is NP-hard. If an instance however is harmonic, things change fundamentally. Recall that an instance is *harmonic*, if for each two tasks of that instance, the period of one task is an integer multiple of the other. We will present a data structure that allows us to design efficient First-Fit type approximation algorithms. We then show that this algorithm is a 2-approximation algorithm. In the following we always assume that tasks $\tau_1, \ldots, \tau_n$ of the input instance $\mathcal{T}$ are sorted such that $p(\tau_j) \mid p(\tau_{j+1})$, for $j = 1, \ldots, n-1$. Moreover, the number of different period lengths of the input-tasks is denoted by k and the set of all task periods is denoted by $Q(\mathcal{T}) = \{q_1, \ldots, q_k\}$, where $q_i \mid q_{i+1}$, for $i = 1, \ldots, k-1$. We first describe the data structure, then present a 2-approximation algorithm, and finally discuss hardness results.

2.3.1 Bin Trees

An important encoding of single-machine schedules which we use repeatedly is the (compressed) *bin tree* that we now explain. Assume that offsets $a(\tau_j)$ for tasks τ_j, $j = 1, \ldots, n$, are given so that they determine a feasible single-machine schedule where no two jobs collide. Observe that the schedule repeats itself after the largest period $q_k = \text{lcm}(q_1, \ldots, q_k)$.

By a simple shifting argument, we can assume that the first task τ_1 with minimum period $p_1 = q_1$ has offset $a_1 = 0$: Observe that differences of offsets stay unchanged when all offsets are shifted by the same number. Hence the algebraic feasibility criterion from Lemma 2.2.4 ensures that a schedule remains feasible when shifting offsets. The smallest period q_1 now partitions the time-horizon $[0, q_k)$ into *bins* $[i \cdot q_1, (i+1) \cdot q_1)$; see Figure 2.2 for an example. Consider two bins $B_i = [i \cdot q_1, (i+1) \cdot q_1)$ and $B_j = [j \cdot q_1, (j+1) \cdot q_1)$ such that $i \equiv j \mod q_r / q_1$. By construction, these intervals are shifted exactly by q_r time units. Hence as far as tasks with period length up to q_r are concerned, these bins look exactly the same. As a shorthand, we write $B_i \equiv_r B_j$ when $i \equiv j \mod q_r / q_1$.

We use this basic fact to describe schedules in a hierarchical structure called a *bin tree* which we now define. The root of a *full bin tree* is a node representing a bin containing all tasks of period length q_1. It has q_2 / q_1 children, each of which represents a bin that contains all its

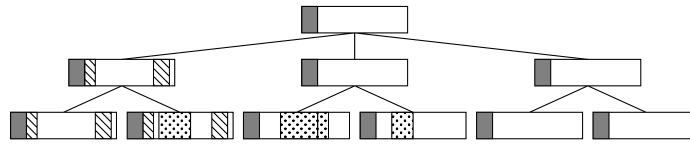

Figure 2.3: The full bin tree corresponding to the schedule in Figure 2.2.

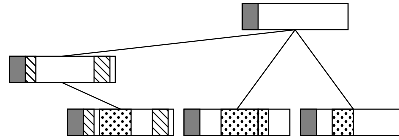

Figure 2.4: The compact form of the bin tree in Figure 2.3.

parent's tasks and may contain additional tasks of period length q_2. We say that the root is of period q_1, and its children are of period q_2. In general, a node B of period q_r contains only tasks of period length up to q_r. If $q_r < q_k$, it has q_{r+1}/q_r children, each of which is a node of period q_{r+1}. Each child of B represents a bin that contains all tasks of B and may contain additional tasks of period length q_{r+1}. Each scheduled task of period length q_r appears in a unique node of period q_r and in all children of that node.

As a consequence of this definition, there is a one-to-one correspondence between nodes of period q_r in the full bin tree and equivalence classes of bins modulo the equivalence relation $\equiv_r$. Furthermore, the hierarchy of equivalence relations $\equiv_r$ $(r = 1,\ldots,k)$ corresponds to the hierarchy of the tree in the following way. If two nodes of period $\geq q_r$ have the same ancestor of period q_r, then their corresponding bins are equivalent modulo $\equiv_r$, and vice versa. In particular the leaves of the bin tree correspond to the bins of the schedule (though not in the same order). Thus we can freely convert between a feasible schedule in terms of task offsets and the corresponding bin tree representation; see Figure 2.3.

The number of nodes in a full bin tree is dominated by its leaves, of which there are q_k/q_1 many. So we cannot operate efficiently on full bin trees and, in particular, we cannot afford to store a full bin tree to implement our First-Fit heuristic. However, if a node of period q_r does not contain a task of period q_r, it is completely determined by its parent. Therefore, we only need to store those nodes of the tree that introduce a new task to the schedule, see Figure 2.4 for an example. In this way we have a *compressed bin tree* whose number of nodes is bounded by the number of tasks and that can be constructed in polynomial time. Coming back to the implementation of First-Fit, given a bin tree and an unscheduled task τ whose period length is greater or equal to the period lengths of all tasks in the bin tree, one can easily determine whether τ can be inserted into the compressed bin tree by checking all the leaves of the compact tree, as well as all nodes to which a new leaf of the right period can be added. We will use this fact in the next subsection.

2.3.2 First-Fit: A 2-Approximation Algorithm

Making use of the bin tree structure and in particular compressed bin trees, one can implement the following First-Fit algorithm efficiently. It maintains a list $M_1, \ldots, M_\ell$ of open machines where M_i was opened before M_j if $i < j$. Each machine is represented by a (compressed) bin tree. The *type* of a machine is the bin size of its tree, i.e., the smallest period of the tasks assigned to the machine. The *remaining capacity* of a bin is the bin size minus the sum of execution times of tasks already assigned to the bin. The algorithm now tries to assign the tasks one by one to the machines, in order of nondecreasing periods. To assign a task τ_j, it considers all machines that have been opened previously (in the order of opening). If the remaining capacity of one of the leaf bins of a machine is at least $c(\tau_j)$, then we assign the task to that bin (ties are broken arbitrarily): pack the task "leftmost" into the bin by assigning the smallest possible offset within the bin so that no collision occurs. If the algorithm fails to assign task τ_j to any of the machines, *two* machines of type $p(\tau_j)$ are opened and the task is assigned to one of them. The fact that two machines instead of one machine is opened is to simplify some averaging arguments in the analysis of the algorithm. See Listing 1 for a more formal description of the algorithm.

Algorithm 1 A 2-approximation algorithm for the periodic maintenance minimization problem with harmonic periods.

Sort instance $\mathcal{T}$ nondecreasing by period length, let $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be in this order.
$\mathcal{M} \leftarrow \{\}$
for $j \leftarrow 1$ **to** n **do**
  for each machine $M \in \mathcal{M}$ (in order of opening) **do**
    if τ_j can be assigned to M **then**
      Assign τ_j to M
      break
    end if
  end for
  if τ_j not assigned **then**
    Open two machines M_1, M_2 of type $p(\tau_j)$
    Assign τ_j to machine M_1.
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{M_1, M_2\}$.
  end if
end for

Note that, because tasks are added in non-decreasing order of period lengths, we only add tasks to leaves of the compressed bin tree. With the discussion from Section 2.3.1, we conclude that the algorithm can be implemented to run in polynomial time. It remains to show that the algorithm has the desired approximation guarantee.

Theorem 2.3.1. *The First-Fit algorithm is a* 2*-approximation algorithm.*

The remainder of this subsection is devoted to the proof of this theorem. In the analysis, for the sake of simplicity, we will always argue via full bin-trees, although the algorithm operates

on compressed bin-trees.

Witnesses

Before we present the proof of Theorem 2.3.1 we discuss some differences to the analysis of the First-Fit algorithm for BIN-PACKING and motivate the concept of a *witness* that turns out to be very useful. The simple observation showing that First-Fit for BIN-PACKING is a 2-approximation is as follows: If First-Fit opens a new bin, then let α be the minimum weight of all previously opened bins. This implies that the current item has weight at least $1-\alpha$ and if there are any other open bins, they must have weight at least $\max\{\alpha, 1-\alpha\}$. The average weight of the bins is thus at least $1/2$.

Now suppose that the First-Fit algorithm for the periodic maintenance minimization problem opens a new machine for task τ_j. Analogous to BIN-PACKING, we would like to argue that the utilization of the open machines (i.e., the total utilization of the tasks assigned to the machine) is high. We then could use the lower bound in terms of utilization from Lemma 2.2.5. Unfortunately this does not work. If the open machines all have a small type, i.e., a bin size less than $c(\tau_j)$, then we have to open a new machine for τ_j regardless of the utilization of the machines. In particular, it is not true that the average utilization of the open machines is at least $1/2$. However, we can derive a lower bound on the average utilization of the machines whose type is *compatible* with τ_j, where the set of compatible types is denoted by $Q(j) := \{q \in Q\colon c(\tau_j) \leq q \leq p(\tau_j)\}$. Hence τ_j serves as a *witness* that those machines are highly utilized on average. Before we state this formally, we provide one useful insight that is helpful for the proof. Recall that another way of characterizing the utilization of a machine is to consider the ratio of time when a machine is busy during a hyper-period (i.e., the least common multiple of the periods) to the total length of the hyper-period. Observe that as the leaf-bins correspond to the bins of the schedule in the first hyper-period, we can equivalently see the utilization as the average *fill ratio* of the leaf-bins. The fill ratio is the ratio of total execution time of tasks in that bin to the bin size.

Lemma 2.3.2. *Consider a schedule computed by the First-Fit algorithm. Let $\tau_j \in \mathcal{T}$ be a task, and let $q \in Q(j)$ be a compatible machine type. Let $M_1, \ldots, M_\ell$ be a set of machines of type q with $\ell \geq 2$. Suppose that τ_j was assigned to a different machine that was opened* after *$M_1, \ldots, M_\ell$.*

Then $\frac{1}{\ell}\sum_{i=1}^{\ell} \mathrm{util}(M_i) > \frac{1}{2}$.

Proof. Consider the situation when the First-Fit algorithm is about to assign task τ_j. The leaf bins of all bin trees from $M_1, \ldots, M_\ell$ are of period at most $p(\tau_j)$. Let α be the minimum fill ratio of all those leaf bins. Clearly $\alpha > 0$ as otherwise, by construction, task τ_j could be added to such a bin. If $\alpha > \frac{1}{2}$, then every bin is more than half filled and the claim follows.

Thus, we can assume that $\alpha \leq \frac{1}{2}$. Let M_i be a machine with a leaf bin that is filled only by α. As $\alpha > 0$, there is a task τ' assigned to that bin. Clearly $c(\tau') \leq q \cdot \alpha$.

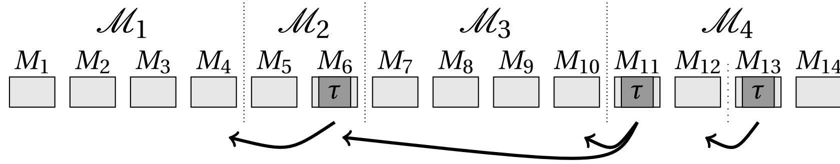

Figure 2.5: The situation of Lemma 2.3.3: Every group of machines but the last has a witness. The last two machines of the last group provide a witness for the rest.

Now for every $k < i$, we claim that the fill ratio of all leaf bins of M_k is more than $1 - \alpha$. This is because otherwise the task τ' could and would have been assigned to machine M_k. On the other hand for $k > i$, if there is a machine M_k and a leaf-bin on that machine that has a fill ratio of at most $1 - \alpha$, then all tasks from that bin could and would have been assigned to machine M_i as there is enough space in the bin of task τ'.

We conclude that for all machines except M_i, every leaf-bin has a fill ratio of more than $1 - \alpha$. Hence the utilization of each of those machines is more than $1 - \alpha$. Every bin on M_i attains at least the minimum fill ratio of α. As $\alpha \leq \frac{1}{2}$, and there are at least two machines, the average utilization of all machine $M_1, \ldots, M_\ell$ more than $\frac{1}{2}$. □

For a period $q \in Q(\mathcal{T})$, let $\mathcal{M}_q$ denote the set of machines of type q. We say that $\mathcal{M}_q$ *has a witness*, if there is a task $\tau \in \mathcal{T}$ compatible to q that is scheduled to a machine of larger type than q. If all machine types have a witness, we can now easily show that we have a 2-approximation.

Lemma 2.3.3. *$FF(\mathcal{T})$ denotes the number of machines opened by the First-Fit heuristic. If for all $q \in Q(\mathcal{T})$, $q < q_k$, the set $\mathcal{M}_q$ has a witness, then $FF(\mathcal{T}) \leq 2 \cdot OPT(\mathcal{T})$.*

Proof. An illustration of the essential proof idea is given in Figure 2.5. Let $q \in Q$ with $q < q_k$, and let τ be a witness for $\mathcal{M}_q$. Then we can apply Lemma 2.3.2 to show that $\sum_{M \in \mathcal{M}_q} \text{util}(M) \geq \frac{1}{2}|\mathcal{M}_q|$. Now let $\mathcal{M}'$ be the set $\mathcal{M}_{q_k}$ without the two last machines that we call $\widetilde{M}_1$ and $\widetilde{M}_2$. Let τ be a task assigned to $\widetilde{M}_1$. Observe that τ is a witness for $\mathcal{M}'$. Thus, $\sum_{M \in \mathcal{M}'} \text{util}(M) \geq \frac{1}{2}|\mathcal{M}'|$. Hence we have $FF(\mathcal{T}) - 2 \leq 2 \cdot \text{util}(\mathcal{T} \setminus \{\tau\})$ which implies $FF(\mathcal{T}) - 2 < 2 \cdot \text{util}(\mathcal{T}) \leq 2 \cdot OPT(\mathcal{T})$. The last inequality is by the utilization bound from Lemma 2.2.5. Since both $FF(I)$ and $2 \cdot OPT(I)$ are even, we conclude $FF(\mathcal{T}) \leq 2 \cdot OPT(I)$. □

If the special case of Lemma 2.3.3 does not apply, we can identify sub-instances that are pairwise independent of each other, yet cover all machines opened by First-Fit. We use this observation to prove Theorem 2.3.1 by induction; see also Figure 2.6.

Proof of Theorem 2.3.1. We prove the theorem by induction on k, the number of different periods. If $k = 1$, then the claim follows directly from Lemma 2.3.3. Now assume that First-Fit

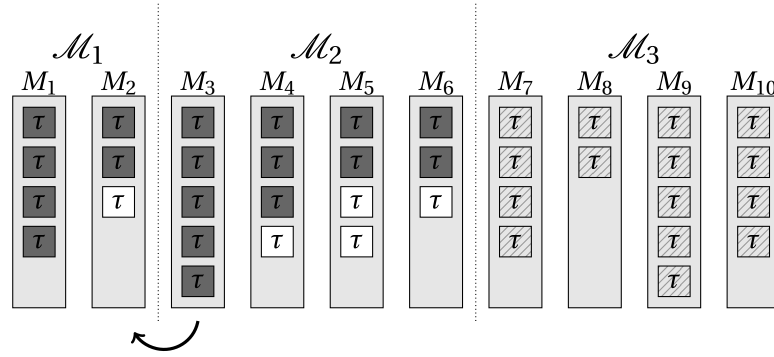

Figure 2.6: Possible situation in the proof of Theorem 2.3.1: Group $\mathcal{M}_2$ does not have a witness. The dark tasks form the set $\overline{\mathcal{T}}$. Together with the white tasks, they form the set $\mathcal{T}'$. The striped tasks form the set $\mathcal{T}''$.

is a 2-approximation for all instances with less than k periods. We want to prove the same for instances with k periods. If for all $q \in Q$, $q < q_k$, the set $\mathcal{M}_q$ has a witness, again the claim follows directly with Lemma 2.3.3.

Thus, let $q \in Q$, $q < q_k$ be a period such that $\mathcal{M}_q$ does not have a witness. We now partition our task set $\mathcal{T}$ as follows.

$$\mathcal{T}' := \{\tau \in \mathcal{T} : \text{ First-Fit assigns } \tau \text{ to a machine of type } \leq q\}$$

and $\mathcal{T}'' := \mathcal{T} \setminus \mathcal{T}'$. Moreover, let $\overline{\mathcal{T}} := \{\tau_i \in \mathcal{T}' : p(\tau_i) \leq q\}$, i.e., the set of tasks assigned to a machine of type $\leq q$, having a period of at most q. Let τ_j be an arbitrary task in $\mathcal{T}''$. Then q is not compatible with τ_j since otherwise τ_j would be a witness for $\mathcal{M}_q$. Thus, each task in $\mathcal{T}''$ has a running time $> q$. As the period lengths of all tasks in $\overline{\mathcal{T}}$ are at most q, no task in $\overline{\mathcal{T}}$ can be scheduled on the same machine as a task in $\mathcal{T}''$. This shows that $\overline{\mathcal{T}}$ and $\mathcal{T}''$ are independent in the sense that $OPT(\overline{\mathcal{T}}) + OPT(\mathcal{T}'') = OPT(\overline{\mathcal{T}} \cup \mathcal{T}'') \leq OPT(\mathcal{T})$.

On the other hand, since First-Fit assigns each task of $\mathcal{T}'$ to a machine of type $\leq q$ and these must have been opened and typed by a task in $\overline{\mathcal{T}}$, one has $FF(\overline{\mathcal{T}}) = FF(\mathcal{T}')$ and $FF(\mathcal{T}) = FF(\overline{\mathcal{T}}) + FF(\mathcal{T}'')$. Using the induction hypothesis, we get

$$FF(\mathcal{T}) = FF(\overline{\mathcal{T}}) + FF(\mathcal{T}'') \leq 2 \cdot OPT(\overline{\mathcal{T}}) + 2 \cdot OPT(\mathcal{T}'') \leq 2 \cdot OPT(\mathcal{T}).$$

This concludes the proof. □

This algorithm can easily be turned into a 2-approximation algorithm for the resource augmentation variant. Consider an instance $\mathcal{J}$ that should be scheduled on m machines. As by assumption for augmented instances, there is a feasible schedule on m speed-one machines, our algorithm from above will compute a feasible schedule for at most $2 \cdot m$ speed-one machines. Observe that for two machines of same type, their schedule *in bin structure* can be

merged into a feasible schedule for one speed-2 machine M as follows: Assume that machine M_1 and M_2 are of type q. Each bin of both machines (and the tasks within) gets scaled down by a factor of 2, i.e., each bin is of length $q/2$. Now for each node in the bin tree(s), we take their scaled bins from M_1 and M_2 and place them into the same node for machine M. We do it in such a way that the scaled bin from machine M_1 occupies the interval $[0, q/2]$, and the scaled bin from M_2 occupies the interval $[q/2, q]$. This way we get a schedule that is feasible for speed-2 machines. We conclude:

Theorem 2.3.4. *There is a 2-approximation algorithm for the resource augmentation variant with harmonic periods.*

2.3.3 Inapproximablity

After the discussion of the 2-approximation algorithm, we now present a matching inapproximability result. This is established with a reduction from PARTITION. The instances generated by the reduction have the harmonic periods property. Consider a PARTITION instance, i.e., n items with sizes $s_1, \ldots, s_n \in \mathbb{N}$. Let $B := \frac{1}{2} \sum_{i=1}^{n} s_i$. The PARTITION-problem is to decide whether or not there is a subset of items whose total size is B. This problem is NP-complete [GJ79]. We will always assume that $B \in \mathbb{N}$, since otherwise the instance is obviously a NO-instance.

We construct an instance for the machine minimization problem with harmonic periods containing $n+1$ tasks $\tau_1, \ldots, \tau_{n+1}$ as follows:

$$c(\tau_j) := s_j,\ p(\tau_j) := 2B + 2\ \forall j = 1, \ldots, n, \quad c(\tau_{n+1}) := 1,\ p(\tau_{n+1}) := B + 1.$$

We define $\mathrm{red}(s_1, \ldots s_n) := \{\tau_1, \ldots, \tau_{n+1}\}$. Clearly, $\mathrm{red}(s_1, \ldots, s_n)$ can be computed from $s_1, \ldots, s_n$ in polynomial time. The intuition behind this construction is as follows. Observe that the constructed tasks have harmonic periods. Therefore, when assigning the tasks to a machine, we can think of assigning them to bins (see Section 2.3.1 for a discussion of the bin concept). We want to assign all tasks to one machine. By construction, task τ_{n+1} defines two bins, each of them having a remaining capacity of B to be filled with the other tasks. Every task τ_j has an execution time of s_j, and thus partitioning the tasks to the two bins essentially encodes the PARTITION problem. Therefore the tasks can be distributed to the two bins if and only if the corresponding PARTITION instance is a YES-instance. A formal proof follows.

Lemma 2.3.5. *Given a* PARTITION *instance* $s_1, \ldots, s_n \in \mathbb{N}$*, the instance is a YES-instance if and only if* $\mathrm{red}(s_1, \ldots, s_n)$ *can be scheduled to one machine.*

Proof. Observe that $\mathrm{util}(\mathrm{red}(s_1, \ldots s_n)) = 1$. Thus if $\mathrm{red}(s_1, \ldots, s_n)$ can be scheduled to one machine, this machine is never idle. Assume that $\mathrm{red}(s_1, \ldots, s_n)$ can be scheduled to one machine, and let $a_1, \ldots, a_{n+1}$ be feasible offsets. Consider the time interval $[a_{n+1} + 1, a_{n+1} + B]$ between the first and the second execution of task τ_{n+1}. Let S be the set of indices of the tasks executed during this interval. Note that every task will be executed at most once during this

interval. Since the processor is never idle, we have

$$B = \sum_{j \in S} s_j,$$

which proves that $s_1, \ldots, s_n$ is a YES-instance.

Now assume that $s_1, \ldots, s_n$ is a YES-instance. Thus there is a subset $S \subseteq [n]$ with

$$B = \sum_{j \in S} s_j.$$

We assign all tasks to one machine. For each $j \in S$, we successively assign task τ_j an offset such that the first execution of these tasks completely occupies the interval $[0, B-1]$. Task τ_{n+1} receives offset B. For all other tasks, we successively assign an offset such that their first execution completely occupies the interval $[B+1, 2B]$. One can easily check that the resulting schedule is feasible. □

This reduction directly implies an inapproximability result:

Corollary 2.3.6. *There is no $(2-\varepsilon)$-approximation algorithm for the machine minimization problem with harmonic periods for any $\varepsilon > 0$, unless $P = NP$.*

Proof. Assume there is $(2-\varepsilon)$-approximation for some $\varepsilon > 0$. Given an instance for the PARTITION problem, Lemma 2.3.5 implies that the algorithm will output a solution using one machine, if it is a YES-instance. It will output a solution using at least two processors otherwise. Thus, this algorithm can be used to decide the NP-complete PARTITION problem. □

Asymptotic approximation

For some APX-hard problems one can get a better approximation guarantee if one allows for asymptotic approximation, i.e., if one searches for algorithms whose approximation guarantee is $c \cdot OPT + o(OPT)$. As mentioned previously, this is true for example for BIN-PACKING. This problem is hard to approximate within a factor of $\frac{3}{2}$. Nevertheless, there is an asymptotic PTAS [FdlVL81]. We now show that this is not true in the case of periodic maintenance minimization. We first describe a construction that will be useful to prove this result. It essentially duplicates instances to "pump up" OPT, but at the same time enforces independent treatment of the duplicates. Let $\mathcal{T}$ be an instance of the machine minimization problem. For some $\ell \in \mathbb{N}$, let $\ell \cdot \mathcal{T}$ denote the instance where each execution time and each period is multiplied by the factor ℓ. Note that there is a one-to-one correspondence between solutions to $\mathcal{T}$ and to $\ell \cdot \mathcal{T}$. Let $B := \max_{\tau_j \in \mathcal{T}} p(\tau_j)$. For any $\ell \geq B$, consider the instance

$$\mathcal{T}' = \mathcal{T} \cup (\ell \cdot \mathcal{T}).$$

Take any two tasks $\tau \in \mathcal{T}$ and $\tau' \in \ell \cdot \mathcal{T}$. Observe that no solution to $\mathcal{T}'$ can assign τ and τ' to the same machine, as the execution time of one is at least as large as the period of the other. Thus the tasks of $\mathcal{T}$ and $\ell \cdot \mathcal{T}$ use disjoint sets of machines, and therefore can be treated individually. This implies that $OPT(\mathcal{T}') = 2 \cdot OPT(\mathcal{T})$. We call this construction a *duplication*, formulated more general as follows: Given an instance $\mathcal{T}$, and $k \in \mathbb{N}$. Let $B := \max_{\tau_j \in \mathcal{T}} p(\tau_j)$. We define

$$\mathrm{dup}(\mathcal{T}, k) := \bigcup_{i=0}^{k-1} B^i \cdot \mathcal{T}.$$

With the preceding discussion, the following is immediate:

Lemma 2.3.7. *For any set of tasks $\mathcal{T}$ and any $k \in \mathbb{N}$, we have*

$$OPT\big(\mathrm{dup}(\mathcal{T}, k)\big) = k \cdot OPT(\mathcal{T}).$$

Also note that the duplication does not destroy the structure of the instances: If $\mathcal{T}$ is an instance with harmonic periods, $\mathrm{dup}(I, B)$ will have harmonic periods as well. We can use the duplication to show that asymptotic approximation algorithms cannot achieve better performance guarantees than non-asymptotic approximation algorithms:

Theorem 2.3.8. *Let c be some constant. Assume that we have a polynomial time algorithm that for any instance $\mathcal{T}$ computes a solution using at most $c \cdot OPT(\mathcal{T}) + o(OPT(\mathcal{T}))$ many machines. Then for any constant $\varepsilon > 0$ there is an $(c + \varepsilon)$-approximation algorithm for the machine minimization problem or the machine minimization problem with harmonic periods, respectively.*

Proof. Assume that we have an algorithm with an $c \cdot OPT + f(OPT)$ approximation guarantee, where $f(OPT) \in o(OPT)$. Then there is a constant M such that: $f(OPT) \leq \varepsilon \cdot OPT$ for all $OPT \geq M$. We construct an $(c+\varepsilon)$-approximation algorithm as follows: Given an instance $\mathcal{T}$, the algorithm duplicates it to obtain the instance $\mathcal{T}' := \mathrm{dup}(\mathcal{T}, M)$. With Lemma 2.3.7 we have $OPT(\mathcal{T}') = M \cdot OPT(\mathcal{T}) \geq M$. Thus the $c \cdot OPT + f(OPT)$ approximation algorithm produces a solution using at most

$$c \cdot OPT(\mathcal{T}') + f(OPT(\mathcal{T}')) \leq (c + \varepsilon) \cdot OPT(\mathcal{T}') = M \cdot (c + \varepsilon) \cdot OPT(\mathcal{T})$$

machines. From the properties of the duplication operation it follows that each copy of $\mathcal{T}$ in the instance $\mathcal{T}'$ must be scheduled to a disjoint subset of machines. Since the total number of machines used is $M \cdot (c+\varepsilon) OPT(\mathcal{T})$, and M copies of $\mathcal{T}$ are assigned to these machines, at least one copy is assigned to at most $(c+\varepsilon) \cdot OPT(\mathcal{T})$ many machines. This is an $(c+\varepsilon)$-approximation for the original instance $\mathcal{T}$. □

As noted before, the duplication does not destroy the structure of harmonic instances. Therefore, this theorem together with Corollary 2.3.6 directly shows that also there is no asymptotic

$(2-\epsilon)$-approximation algorithm.

Theorem 2.3.9. *There is no $(2-\varepsilon)OPT + o(OPT)$ approximation algorithm for the machine minimization problem with harmonic periods for any $\varepsilon > 0$, unless $P = NP$.*

2.4 Further Research Directions

For both the periodic maintenance minimization problem and the special case of harmonic periods, we have a complete characterization of the approximation landscape with tight lower and upper bounds on the approximation factor (under the assumption that $P \neq NP$). We have seen however that there is a drastic difference in the algorithmic complexity of the periodic maintenance problem compared to the special case with harmonic periods. Where the general problem is inapproximable within almost every nontrivial factor, there is a 2-approximation algorithm for the harmonic sub-case. This rises the question of whether there is something "in between" these two extreme cases. Are there sub-cases more general than harmonic periods that still admit nontrivial approximation algorithms? One way to characterize these generalizations is via the *divisibility graph* of the periods. The divisibility graph is a directed graph that has a node $v \in V$ for every different period of the problem instance. Two nodes u and v share an arc (u, v) if the period of u divides the other, and there is no "intermediate node", i.e., no node w so that the period of u divides the period of w and the period of w divides the period of v. Observe that if the periods are harmonic, this graph is simply a path. A generalization would be to consider instances whose divisibility graphs form an out-tree (i.e., a tree where each node except for the root has in-degree one). The concept of bin-trees can be generalized to this setting in a more or less natural way: a bin of the bin-tree corresponding to a period that has more than one descendant in the divisibility graph has to partition the remaining free space in the bin amongst these descendants. For each descendant in the divisibility tree, the bin-node in the bin tree would then have the usual amount of child bins for that period, and all of these children can use only the space reserved for them in the partition. How to design a good first-fit heuristic however becomes less obvious. Another way to generalize the harmonic case would be via partial orders: Observe that the divisibility of the periods defines a partial order. It is a complete order if and only if the periods are harmonic. One could study certain classes of partial orders and see whether they admit non-trivial approximation algorithms.

3 Real-time Avionics Optimization in Practice

3.1 Introduction

In Chapter 2, we studied the periodic maintenance minimization problem in a mathematically rigorous way, leading to a characterization of the complexity landscape of the problem. Our industrial partner however is not so much interested in the theoretical worst-case complexity of the problem, but rather in a solution to his concrete problem instances that should be found in a "reasonable" amount of time. Moreover, the problem as it appears in the real world is more complex than the "pure" periodic maintenance minimization problem: Tasks are not only characterized by an execution time/period pair, but also have additional requirements in terms of memory, I/O, operational availability, functional separation, and functional grouping. The processors have limited memory capabilities and limited I/O capabilities via a network.

For that reason, to address the problem on real-world instances and in their full complexity, we rely on integer programming approaches. An integer program is a linear program whose variables are constrained to be integer. Integer programming is a very powerful concept which allows to model a huge range of combinatorial optimization problems. Unfortunately this implies that solving integer programs is NP-hard. Nevertheless, there exists a rich spectrum of methods and tools to solve integer programs for practical purposes. As the problem is NP-hard, the focus for these methods is not to have an algorithm of polynomial worst-case complexity, but an algorithm that runs "fast" on "many" instances of "moderate" size. Developing good methods is a huge and active research area on its own, but not the focus of this work. We refer to a book by Laurence A. Wolsey [Wol98] for an extensive introduction into the subject. Although these methods get increasingly powerful, they are yet far from solving all practically relevant integer programming formulations, even of moderate size. For that reason, it is not sufficient to only find *some* integer programming model for a combinatorial optimization problem, as the efficiency of the solvers running on different models of the same problem can vary drastically. This motivates to develop good integer programming formulations for the periodic maintenance problem, which will be the focus of this chapter.

It turns out that textbook formulations, like time-indexed formulations, are not even remotely

strong enough to tackle problem instances of real-world size. However by exploiting structural insights of the problem we provide integer programming models that outperform the textbook approach by orders of magnitude. Our first formulation is based on the algebraic feasibility criterion from Lemma 2.2.4. Linearizing this criterion we obtain a compact integer programming model. Although the performance of this model is much better than the basic textbook approach, it is still not sufficient to solve industrial size instances. As the real-world instances however are harmonic, we can rely on the theoretical insights from Chapter 2 regarding harmonic instances: The structure of *bin trees* allows us to develop a fundamentally different integer programming model that is able to handle industrial size instances of the scheduling problem. Even for real-world instances not belonging to the subclass, one can still use the formulation as a heuristic using a rounding technique.

3.1.1 Problem Definition

We quickly recap the formal definition of the *periodic maintenance minimization problem* from Chapter 2. We are given a set of tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ where each task τ_j is characterized by its *execution time* $c(\tau_j) \in \mathbb{N}$ and *period* $p(\tau_j) \in \mathbb{N}$. The goal is to assign the tasks to a set of machines and to compute offsets $a(\tau_j) \in \mathbb{N}_0$. A task τ_j generates one *job* with execution time $c(\tau_j)$ at every time unit $a(\tau_j) + p(\tau_j) \cdot k$ for all $k \in \mathbb{N}_0$. Each job needs to be processed immediately and non-preemptively after its generation on the task's machine. A *collision* occurs if two jobs are simultaneously active on the same machine. A schedule is feasible if no collision occurs. The problem is to find an assignment of the tasks to a minimum number of machines, so that for each machine, a feasible schedule can be found. We now call this problem the *basic periodic maintenance problem* (basic PMP). We denote by $Q = \{q_1, ..., q_k\}$ the set of all period lengths arising in the respective instance. We assume that $q_1 < \ldots < q_k$. An instance has *harmonic periods* if for each pair of tasks τ_i, τ_j we have that either $p(\tau_i) | p(\tau_j)$ or $p(\tau_j) | p(\tau_i)$.

In the *extended PMP*, machines have additional resource limitations in terms of memory of different types (RAM, ROM, etc.) and communication links that need to be considered. Each task has a given requirement for each type of memory and needs certain communication links to be open on its machine. Each machine can handle only a limited number of links and has a limit on the total bandwidth used by them. Moreover, due to system stability requirements and operation functionality, certain tasks need to be assigned to different machines, where certain other tasks have to be assigned to the same machine. Also, the machines are partitioned into two cabinets (left and right). Some of the tasks have to be distributed evenly among the cabinets in order to design a fail-safe architecture.

3.1.2 Related Work

See Section 2.1.1 for a review of previous theoretical work on the periodic maintenance minimization and related problems. Here we discuss literature regarding computational work.

To our best knowledge, we were the first to do computational studies on this problem. After our results where published in [EKM+10], Al Scheikh, Brun, Hladik and Prabhu followed with further computational analysis. In [ASBH10] they propose an integer programming formulation very similar to our *congruence-formulation* that we will explain below. In [ASBHP11] they propose an iterative procedure to solve the problem.

In contrast to the sparse literature concerning computational studies of the periodic maintenance problem, for the special case of the BIN-PACKING problem, there exists extensive literature on heuristics as well as branch-and-bound and column generations methods. In [MT90] polynomial time approximation algorithms as well as lower bounds, and exact algorithms are studied. Lower bounds to the optimal solution (which can be computed fast) are presented in [CPPT07]. Several approaches have been proposed for solving the BIN-PACKING problem with branch-and-price techniques; see, e.g., [Van99, VBJN94, VdC99]. An algorithm called BISON is proposed in [SKJ97], where branch-and-bound techniques and tabu search are combined to design an exact hybrid algorithm.

Regarding heuristics, there is also a lot of literature for the BIN-PACKING problem. For example, Gupta and Ho propose a heuristic that greedily minimizes the slack of the machines. Fleszar and Hindi [FH02] modify these ideas and combine them with variable neighborhood search to design an hybrid algorithm. Moreover, Loh et al. [LGW08] propose a simple local heuristic based in the concept of *weighted annealing*.

3.1.3 Our Contribution

The here presented methods lead, for the first time, to an industry strength tool to optimally schedule real-word problems from the avionics industry. This is achieved by the following steps.

- We describe two integer programming models that differ considerably from the textbook time-indexed formulation. Both formulations outperform the time-indexed formulation.

- The two models have orthogonal strengths and weaknesses: While the *congruence formulation* is of polynomial size, the *bin-formulation* is pseudopolynomial. Furthermore the bin-formulation applies only to *harmonic instances* but outperforms the congruence formulation by several orders of magnitude.

- Many real-world instances are “almost harmonic” which means that only few tasks need to be removed from the instance to make it harmonic. The industry strength method to solve difficult instances is based on *rounding* these outlying periods to the nearest smaller period in the total order.

3.1.4 Outline

This chapter is organized as follows. In Section 3.2, we present two IP-formulations for the basic PMP: the textbook time-indexed approach and a formulation based on linear congruences and the algebraic feasibility criterion. Then in Section 3.3, we discuss the case of harmonic PMP and explain the IP formulation based on bin trees. Afterwards in Section 3.4, we describe how to cope with the additional constraints needed for the extended PMP. Finally, our benchmark results are given Section in 3.5: We compare the three IP-formulation as well as the results obtained by a (greedy) first-fit heuristic.

3.2 General IP-Formulations

As indicated before, we now describe two integer programming formulations for the basic PMP. First we consider a *time indexed formulation*, where we have variables assigning tasks to time slots on each machine. Then, we present a less naïve approach that exploits the algebraic feasibility criterion of Lemma 2.2.4. We call this model the *congruence-formulation*. It uses variables that indicate the offset of each task. The formulations presented in this section are formulated as “feasibility tests”: For a given number of machines m, they compute a feasible assignment to the machines or assert that no such assignment exists. All formulations can easily be modified to model the machine minimization variant using standard techniques. We chose the feasibility variant here to keep the presentation as simple as possible.

3.2.1 Time-Indexed-Formulation

Our first formulation is a naïve formulation that uses binary variables to indicate the offset and machine assignment for each task. More precisely, we consider variables $w_{j,i,t} \in \{0,1\}$ for each $\tau_j \in \mathcal{T}$, $i \in [m]$ and $t \in \{0,1,\dots,p(\tau_j)\}$. If $w_{j,i,t}$ is set to one, this indicates that task τ_j should be assigned to machine i with an offset of t. Otherwise it should be zero. The integer program then looks as follows:

$$\sum_{i\in[m]} \sum_{t\in\{0,1,\dots,p(\tau_j)-1\}} w_{j,i,t} = 1 \qquad \forall \tau_j \in \mathcal{T} \tag{3.1}$$

$$w_{j,i,t} + w_{j',i,t'} \leq 1 \qquad \forall \tau_j, \tau_{j'} \in \mathcal{T}\, \forall i \in [m] \forall t, t' \text{ leading to collision} \tag{3.2}$$

$$w_{j,i,t} \in \{0,1\} \qquad \forall \tau_j \in \mathcal{T}\, \forall i \in [m] \forall t$$

The constraints of type (3.1) make sure that every task gets assigned exactly one machine and offset. The constraints of type (3.2) ensure that the assignment computed by the integer program is feasible. We conclude that if there is a feasible schedule on m machines for instance $\mathcal{T}$, the integer program will compute one. However, the efficiency of this IP is extremely poor. The total number of variables is is $\Theta(|\mathcal{T}|\cdot m\cdot q_k)$ and the number of constraints is $\Theta(|\mathcal{T}|^2\cdot m\cdot q_k^2)$. This is exponential in the encoding size of the input, and also huge for real world instances. It is little surprising that this very naive IP formulation performs very bad in practice, however

this is the only approach “from the book” that was previously known for this problem.

3.2.2 Congruence-Formulation

Now we describe our congruence-formulation for the basic PMP. Unlike the time-indexed formulation, its complexity is polynomial in the input size. The main idea is to use a linearization of the algebraic feasibility criterion from Lemma 2.2.4 to model offset feasibility. To this end, we introduce integer variables a_j for each task $\tau_j \in \mathcal{T}$, which models the offset for task τ_j. In order to check whether two tasks collide we linearize the feasibility criterion from Lemma 2.2.4: Two tasks τ_j and $\tau_{j'}$ are collision free if and only if

$$c(\tau_j) \le a_{j'} - a_j + s_{j,j'} \cdot \gcd(p(\tau_j), p(\tau'_j)) \le \gcd(p(\tau_j), p(\tau'_j)) - c(\tau_{j'}),$$

where $s_{j,j'} \in \mathbb{Z}$ is an *integer variable.*

The machine assignment is done via additional binary variables $x_{j,i} \in \{0,1\}$ for each machine $i \in [m]$, indicating whether or not task j is assigned to machine i. The full IP model looks as follows. As shorthand we set $\gcd_{j,j'} := \gcd(p(\tau_j), p(\tau'_j))$.

$$\sum_{i \in [m]} x_{j,i} = 1 \qquad \forall \tau_j \in \mathcal{T} \tag{3.3}$$

$$y_{j,j'} \ge x_{j,i} + x_{j',i} - 1 \qquad \forall \tau_j, \tau_{j'} \in \mathcal{T}\ \forall i \in [m] \tag{3.4}$$

$$y_{j,j'} \cdot c(\tau_j) \le a_{j'} - a_j + s_{j,j'} \cdot \gcd_{j,j'} \qquad \forall \tau_j, \tau_{j'} \in \mathcal{T} \tag{3.5}$$

$$\gcd_{j,j'} - y_{j,j'} \cdot c(\tau_{j'}) \ge a_{j'} - a_j + s_{j,j'} \cdot \gcd_{j,j'} \qquad \forall \tau_j, \tau_{j'} \in \mathcal{T} \tag{3.6}$$

$$x_{j,i} \in \{0,1\} \qquad \forall \tau_j \in \mathcal{T}\ \forall i \in [m]$$

$$s_{j,j'} \in \mathbb{Z} \qquad \forall \tau_j, \tau_{j'} \in \mathcal{T}$$

The equations of type (3.3) make sure that every task gets assigned to exactly one machine. In the inequalities of type (3.4) we use variables $y_{j,j'}$ indicating whether two tasks τ_j and $\tau_{j'}$ are assigned to the same machine. Constraints (3.4) make sure that if τ_j and τ'_j are assigned to the same machine, $y_{j,j'}$ is (at least) one. Otherwise it can be 0. Finally constraints (3.5) and (3.6) ensure that the offsets computed for tasks assigned to the same machine are collision free: if $y_{j,j'}$ is forced to one, these constraints are exactly the linearization of the algebraic feasibility criterion as discussed above. Otherwise, if $y_{j,j'}$ is 0, the constraint is trivially satisfiable: for any choice of a_j and $a_{j'}$ there is an integer $s_{j,j'}$ that makes the constraints (3.5) and (3.6) feasible. This way the collision constraint is disabled for tasks assigned to different machines. We conclude that if a feasible m machine solution to the problem exists, our formulation will find one. In total, we have $\Theta(|\mathcal{T}|^2 + |\mathcal{T}| \cdot m)$ variables and $\Theta(|\mathcal{T}|^2 \cdot m)$ constraints. The size of the formulation is thus polynomial in the input size.

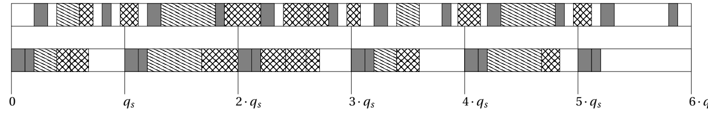

Figure 3.1: A schedule for a single machine and a schedule for the same tasks which is in bin structure. The gray jobs belong to tasks with period length q_1, the striped jobs to tasks with period length $q_2 = 3 \cdot q_1$, and the checkered jobs to tasks with period length $q_3 = 6 \cdot q_1$.

3.3 Harmonic IP-Formulation

We now consider the easier case of instances with harmonic periods. Before we state our IP-formulation tailored to this special case, we discuss the deeper structural insights necessary to get to this formulation.

3.3.1 Structural Insights

Consider a harmonic instance $\mathcal{T}$, and recall the concept of (full) bin trees as introduced in Section 2.3.1: A feasible schedule for a single machine can be represented by a tree of hierarchical bins. The tree has a level for each period length $q_1, \ldots, q_k$. On level q_r, there are q_r / q_1 many bin-nodes, each representing an equivalence class w.r.t. $\equiv_r$, and containing only tasks with period at most q_r. Only tasks of period q_r are new, the others are inherited from the ancestors.

In Section 2.3.1 we have seen that feasible schedules can always be expressed in terms of bin trees, which turned out to be very useful when designing First-Fit type algorithms. For the IP formulation however, we need even more structure. For a task τ assigned to a bin of the tree, we define its *relative offset* as the offset within the bin, i.e., the distance from the start of the bin. We say that a schedule *is in bin structure*, if the following two conditions hold for every bin of the corresponding tree:

- Tasks are ordered non-decreasingly by period length, i.e., there is no task of larger period length that has smaller relative offset than a task of smaller period length in the same bin.
- There are no idle times in a bin between task executions: If a bin is idle at time t, then all relative offsets of tasks in the bin are smaller than t.

Figure 3.1 depicts a feasible single machine schedule together with a schedule for the same instance in bin structure. Figure 3.2 represents the latter schedule as a bin tree.

It is true that if an instance $\mathcal{T}$ has a feasible single machine schedule, then it also has one in bin structure. This is proven indirectly with the following lemma. It allows us to completely reformulate the problem of computing a schedule. Rather than computing offsets that are

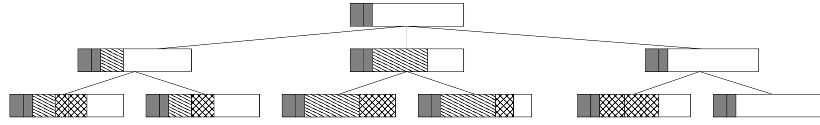

Figure 3.2: The bin tree corresponding to the schedule in Figure 3.1.

non-colliding, we formulate it as a problem of assigning tasks to bins in such a way that no bin is "over-packed".

Lemma 3.3.1. *Consider an instance $\mathcal{T}$ with period lengths $q_1 < \ldots < q_k$. There is a feasible single-machine schedule for $\mathcal{T}$ if and only if the following assignment problem has a solution. For each task $\tau \in \mathcal{T}$, assign it to one of the bins of period $p(\tau)$ in the bin tree, and all its descendants. The assignment must be in such a way that no bin is over-packed, i.e., the total execution time of tasks assigned to each bin is at most q_1.*

Proof. Clearly, if there is a feasible schedule, its bin tree gives rise to a solution to the assignment problem in the straightforward way. Now lets assume that we have a solution to the assignment problem, and want to convert it into a feasible schedule. This is done with a greedy algorithm: Sort the tasks non-decreasingly by period length. Then assign them to the bins as indicated in the solution to the assignment problem, assigning them the smallest possible relative offset. The fact that no bin is over-packed in the bin assignment problem makes sure that we can always do that. Observe that this way we always obtain a feasible schedule in bin structure. □

3.3.2 Bin-Formulation

Lemma 3.3.1 allows us to reinterpret the scheduling problem as a bin-assignment problem. First for simplicity assume that we want to assign all tasks to a single machine. Lemma 3.3.1 tells us that we only need to find an assignment of tasks to corresponding bins in the bin tree so that no bin is overloaded. A task $\tau_j \in \mathcal{T}$ has $p(\tau_j)/q_1$ many choices for bins to assign to, i.e., the bins on the tree level of period $p(\tau_j)$. We model this by introducing variables $z_{j,\ell} \in \{0,1\}$ for $\ell = 0,\ldots,p(\tau_j)/q_1 - 1$. Variable $z_{j,\ell}$ is 1 if and only if task j is assigned to bin ℓ. To make sure that every task gets assigned to a bin, we add constraints of the form

$$\sum_{\ell=0}^{p(\tau_j)/q_1-1} z_{j,\ell} = 1 \qquad \forall \tau_j \in \mathcal{T}.$$

Now we model the packing problem. If τ_j is assigned to bin ℓ-th bin on level $p(\tau_j)$ of the bin tree, it is automatically assigned also to all its descendants. Leaf bins that are descendants of that bin are precisely the bins $B_{\ell'}$ so that $\ell' \equiv \ell \mod p(\tau_j)/q_1$. Equivalently, task τ_j appears in bin $B_{\ell'}$ if and only if $z_{j,(\ell' \bmod p(\tau_j)/q_1)} = 1$. Hence we can guarantee that a leaf bin is not

overloaded by imposing a knapsack type constraint:

$$\sum_{\tau_j \in \mathcal{T}} c(\tau_j) \cdot z_{j,(\ell \bmod p(\tau_j)/q_1)} \leq q_1 \qquad \forall \ell \in \{0,\ldots,q_k/q_1 - 1\}.$$

It is sufficient to pose this constraint for the leaf bins: for every non-leaf bin there is a leaf bin containing a super-set of tasks. Summarizing, we can model the single machine basic PMP for harmonic periods as follows:

$$\begin{aligned}
\sum_{\ell=0}^{p(\tau_j)/q_1-1} z_{j,\ell} &= 1 && \forall \tau_j \in \mathcal{T} \\
\sum_{\tau_j \in \mathcal{T}} c(\tau_j) \cdot z_{j,(\ell \bmod p(\tau_j)/q_1)} &\leq q_1 && \forall \ell \in \{0,\ldots,q_k/q_1 - 1\} \\
z_{j,\ell} &\in \{0,1\} && \forall \tau_j \in \mathcal{T}\ \forall \ell \in 0,1,\ldots,p(\tau_j)/q_1 - 1.
\end{aligned}$$

We now turn to the multi-machine PMP. Analogous to the single machine case, we consider variables $z_{j,i,\ell} \in \{0,1\}$ that indicates whether a task $\tau_j \in \mathcal{T}$ is assigned to the ℓ-th bin on machine i for $\ell \in \{0,\ldots,p(\tau_j)/q_1 - 1\}$. An extra difficulty we now have to face is that we do not know a priori which is the smallest period length appearing on each machine. In particular we do not know the bin size of the machine. We overcome this by simultaneously modeling all bin sizes. For every period q_r, $r = 1,\ldots,k$, we introduce a variable $z^r_{j,i,\ell} \in \{0,1\}$. It is valid only if the bin-size is q_r and indicates whether a task $\tau_j \in \mathcal{T}$ is assigned to the ℓ-th bin on machine i for $\ell \in \{0,\ldots,p(\tau_j)/q_r - 1\}$. It is used only if the smallest period assigned to machine i is q_r. The corresponding bin packing constraint is then

$$\sum_{\tau_j \in \mathcal{T}} c(\tau_j) \cdot z^r_{j,i,(\ell \bmod p(\tau_j)/q_r)} \leq q_r \qquad \forall i \in [m], \forall \ell \in \left\{0,\ldots,\frac{q_k}{q_r} - 1\right\}, \forall r = 1,\ldots,k.$$

We still need to disable these packing constraints in case that no task of that period is assigned to the machine. To this end, for each period length q_r and each machine $i \in [m]$ we introduce an indicator variable $d_{i,r} \in \{0,1\}$. It is 1 if and only if a task of period q_r is assigned to machine i. This variable is modeled by constraints of type

$$d_{i,r} \geq \sum_{\ell=0}^{p(\tau_j)/q_1-1} z_{j,i,\ell} \qquad \forall i \in [m], \forall r \in \{1,\ldots,k\}, \forall \tau_j : p(\tau_j) = q_r$$

The packing constraint is then modified to

$$\sum_{\tau_j \in \mathcal{T}} c(\tau_j) \cdot z^r_{j,i,\ell} \leq q_r + (1 - d_{i,r}) q_k \qquad \forall i \in [m], \forall \ell \in \left\{0,\ldots,\frac{q_k}{q_r} - 1\right\}, \forall r = 1,\ldots,k$$

and is therefore disabled if there is no task of period q_r. Finally we model the $z^r_{j,i,\ell}$ by “glueing”

neighboring q_1 sized bins. We set:

$$z^r_{j,i,\ell} = \sum_{\ell'=\ell\cdot q_r/q_1}^{(\ell+1)\cdot q_r/q_1-1} z_{j,i,\ell'} \qquad \forall \tau_j, \forall i \in [m], \forall r \in \{1,\ldots,k\}, \forall \ell \in \{0,\ldots,q_r/q_1-1\}.$$

In summary, the bin formulation looks as follows:

$$\begin{aligned}
\sum_{i\in[m]} \sum_{\ell=0}^{p(\tau_j)/q_1-1} z_{j,i,\ell} &= 1 && \forall \tau_j \in \mathcal{T}\\
z^r_{j,i,\ell} &= \sum_{\ell'=\ell\cdot q_r/q_1}^{(\ell+1)\cdot q_r/q_1-1} z_{j,i,\ell'} && \forall \tau_j, \forall i, \forall r \in \{1,\ldots,k\}, \forall \ell \in \{0,\ldots,q_r/q_1-1\}\\
d_{i,r} &\geq \sum_{\ell=0}^{p(\tau_j)/q_1-1} z_{j,i,\ell} && \forall i \in [m], \forall r \in \{1,\ldots,k\}, \forall \tau_j : p(\tau_j) = q_r\\
\sum_{\tau_j\in\mathcal{T}} c(\tau_j)\cdot z^r_{j,i,\ell} &\leq q_r + (1-d_{i,r})q_k && \forall i \in [m], \forall \ell \in \left\{0,\ldots,\frac{q_k}{q_r}-1\right\}, \forall r = 1,\ldots,k\\
z_{j,i,\ell} &\in \{0,1\}\\
z^r_{j,i,\ell} &\in \{0,1\}.
\end{aligned}$$

Note that we actually do not need the variables $z^r_{j,i,\ell}$ explicitly: We can replace each occurrence by a sum from the second line of the model from above. This reduces the complexity a bit. After this replacement, the bin-formulation needs $\Theta\left(|\mathcal{T}|\cdot m\cdot \frac{q_k}{q_1}\right)$ variables and constraints in total. This is pseudopolynomial in the input. However the ratio q_k/q_1 usually is rather small for instances from the real world, so that the total size of this model for these instances is rather small.

3.4 Extended Constraints

We now briefly describe how to incorporate the additional requirements of the extended PMP into our IP models. This adaption is quite straightforward, simply by adding some additional constraints. These constraints however require binary variables $x_{j,i} \in \{0,1\}$ indicating whether or not task τ_j is assigned to machine i. We also need binary variables $y_{j,j'} \in \{0,1\}$ that are one if tasks τ_j and $\tau_{j'}$ are assigned to the same machine. These variables are already part of the congruence formulation. For the other formulations they can be modeled analogously. We skip the details.

The extended PMP has several types of memory constraints: Every processor has limited memory, each task has a certain memory requirement, and a feasible assignment of tasks to processors must ensure that the sum of memory requirements of tasks assigned to a processor do not exceed its memory limits. These constraints are modeled with knapsack inequalities of

the form

$$\sum_{\tau_j \in \mathcal{T}} m_j x_{j,i} \leq M_i \quad \forall i \in [m],$$

here m_j and M_i denote the memory requirements and limits, respectively. Also, each machine can open a set of communication links. Denote by S the set of all possible links. Each link $s \in S$ needs a bandwidth b_s. For each machine, the total number of open links is bounded by a value TL and the total bandwidth available for links is bounded by a value TB. For each task τ_j we are given a set $S_j \subseteq S$ of links which are required to be opened on a machine which τ_j is assigned to. A link is opened only once even if used by several tasks on the same processor. We introduce binary variables $o_{s,i} \in \{0,1\}$ which indicate whether link s is open on machine M_i. The right behavior can easily be modeled analogous to what we did before. Using these variables, we ensure that the bounds in terms of bandwidth and number of links is not violated by adding more knapsack type constraints. Next, there are certain pairs of tasks which need to be scheduled on the same machine or on different machines. We can force tasks $\tau_j, \tau_{j'}$ to be assigned to different machines by setting $y_{j,j'} = 0$, where y is the variable indicating that two tasks are on the same machine as defined above. We can ensure that the tasks are assigned to the same machine by setting $x_{j,i} = x_{j',i}$ for all machines i. Finally, there are sets of tasks $\mathcal{T}' \subseteq \mathcal{T}$ which have to be distributed evenly among the left and the right cabinet of machines. Formally, it is required that $|\mathcal{T}'|/2$ many tasks of them are scheduled on machines in the left and in the right cabinet if $|\mathcal{T}'|$ is even. If $|\mathcal{T}'|$ is odd, it is required that $(|\mathcal{T}'|+1)/2$ tasks are assigned to the left cabinet and $(|\mathcal{T}'|-1)/2$ tasks are assigned to the right cabinet. For each machine M_i, it is pre-defined whether it is located in the left or in the right cabinet. This partitions the set of machines $\mathcal{M}$ into sets $\mathcal{M}_L$ and $\mathcal{M}_R$. Now for all these "special" sets $\mathcal{T}'$ we ensure these balancing requirements by adding the following constraints.

$$\sum_{j \in \mathcal{T}'} \sum_{M_i \in \mathcal{M}_L} x_{i,j} = \sum_{j \in \mathcal{T}'} \sum_{M_i \in \mathcal{M}_R} x_{i,j} \qquad \forall \text{"special" sets } \mathcal{T}' \text{ with even } |T'|$$

$$\sum_{j \in \mathcal{T}'} \sum_{M_i \in \mathcal{M}_L} x_{i,j} = 1 + \sum_{j \in \mathcal{T}'} \sum_{M_i \in \mathcal{M}_R} x_{i,j} \qquad \forall \text{"special" sets } \mathcal{T}' \text{ with odd } |T'|$$

3.5 Computational Results

In this section we present our computational results. We solved all real-world instances provided by our industrial partner within minutes, with all constraints of the extended PMP and some additional constraints that we describe later. The most difficult instance has 177 tasks and needs 16 machines. The period lengths are almost harmonic (see details below). Instances of this size are far beyond of what the time-indexed formulation and the congruence formulation can solve in a reasonable amount of time. However, the special design of the bin-formulation allowed to solve the instances within 15 minutes.

For benchmarking purposes, we first analyze how our different models perform on random instances. Moreover, we study the quality of solutions obtained by a First-Fit heuristic. The heuristic orders the tasks by period length and execution time and greedily assigns them

to the first machine where it can find a suitable start offset. Note that for the basic PMP in the harmonic case, this is already a 2-approximation algorithm (see Section 2.3.2). In the non-harmonic case, the algorithm tries all possible (integer) offsets and is thus only of pseudopolynomial running time. Reflecting the theoretical results, our benchmarking shows that First-Fit has a good performance in the basic PMP. However, as we will see, it does not cope well with the additional constraints of the extended PMP.

Due to the novelty of our problem, there is no existing standard set of instances for benchmarking. Therefore, we must rely on generating random instances. We consider two ways of generating random instances: purely random instances and random perturbations of real-world instances arising at our industrial partner. We will call the latter instances the real-world perturbed (RWP) instances. There are four different settings: for the basic PMP, we consider the non-harmonic case with purely random instances, the harmonic case with purely random instances, and the harmonic case with RWP instances. For the extended PMP, we benchmark only with RWP instances in the harmonic case. All computations were done on a two-processor machine with Intel Xeon 2.66 GHz CPUs with 8 GB of RAM, running Linux. We used CPLEX release version 12.1.0.

We remark that additionally we introduce some cuts to the IP formulations. If for two tasks, the sum of their execution times exceeds the greatest common divisor of their periods, Lemma 2.2.4 implies that they cannot be assigned to the same machine. Thus we can add separation constraints for these tasks similar to those used in the extended PMP model. Moreover the utilization bound from Lemma 2.2.5 provides knapsack type constraints. Notice that in all our IP-formulations we need an upper bound on the number of machines. This was obtained by first running the First-Fit heuristic.

Non-harmonic Case

In the non-harmonic case we benchmark the following IP-formulations and algorithms: the time-indexed-formulation (TIF), the congruence-formulation (CF), and the First-Fit heuristic (FF). For each purely random instance we drew five different period lengths from the set $\left\{2^x \cdot 3^y \cdot 50 \mid x \in \{0,\dots,4\}, y \in \{0,\dots,3\}\right\}$ uniformly at random. This is a typical number of period lengths in real-world instances. For each task τ_j, its period length $p(\tau_j)$ is chosen uniformly at random from one of the five period lengths. (In our experiments we observed that larger values for the number of period lengths in an instance result in instances which are harder to solve; however, the relation of the running times between the three IP-formulations remains the same.) Its execution time is drawn from an inverse exponential distribution, i.e., $c(\tau_i) = p(\tau_j) \cdot e^{-\ln p(\tau_j) \cdot x}$. This results in realistically small execution times in comparison with the period length and hence mimics the real instances from our industrial partner. We created 200 random instances each for the case of 10, 20, 30, 40, and 50 tasks. Whenever ten runs in a row did not finish before the timeout of 30 minutes or ran out of memory, we did not consider the respective formulation any further (denoted by dashes in the table).

	IP-formulations				Heuristic
# tasks	CF		TIF		FF
10	0.11s	98%	–	0%	2.99%
20	2.52s	92%	–	0%	2.23%
30	277.41s	42%	–	0%	1.92%

Table 3.1: Computational results for the purely random instances in the non-harmonic case.

Table 3.1 shows our computational results. In all our tables, for each IP-formulation the left column shows the average running times in seconds[1]. The right column shows the percentage of instances that could be solved to optimality within the time limit. For the First-Fit algorithm, we show the average relative error (in %) of the solutions with respect to the optimal solution. The running time of First-Fit is negligible.

Discussion. The First-Fit heuristic apparently performs very well, obtaining the optimal solution most of the time regardless of the number of tasks. This is somewhat surprising given that the problem is almost intractable in terms of approximation algorithms (i.e., NP-hard to approximate within a factor of $|\mathscr{T}|^{1-\varepsilon}$), see Chapter 2. However, the instances created in that reduction are very special and not likely to arise in our random draws. We notice that TIF is impractical even for small instances due to the huge number of integer variables involved in the formulation. In comparison, CF does much better and is able to solve most instances with up to 30 tasks in reasonable time (less than 30 min.).

Harmonic Case

In the harmonic case we benchmark the following IP-formulations/algorithms: the time-indexed-formulation (TIF), the congruence-formulation (CF), the bin-formulation (BF) and the First-Fit heuristic (FF). In the harmonic case, the purely random instances were created by first generating a harmonic sequence of five periods in the following way: We start with period length 50 and successively generate the other periods by multiplying two, three, or six to the previous period. The periods and execution times for the tasks are drawn as in the non-harmonic case. The RWP instances were created by taking tasks uniformly at random from a large harmonic instance from our industrial partner, and perturbing execution time and – for the extended PMP – the memory requirements randomly by up to 25 %. The other extended constraints remain unchanged.

For the purely random instance we consider the basic PMP only. When running the RWP instances we consider both the basic and the extended PMP. Tables 3.2, 3.3, and 3.4 show our computational results for the harmonic case. For the IP-formulations the value in parenthesis denotes the ratio between the respective running time and the time needed by the bin-

[1] We use the shifted geometric mean of running times t_i calculated by $\left(\prod_{i=1}^{n}(t_i+1)\right)^{1/n}-1$. We use the shift in order to decrease the strong influence of the very easy instances in the mean values.

	IP-formulations						FF
# tasks	BF		CF		TIF		FF
10	0.28s (1×)	99%	0.25s (0.9×)	97%	–	0%	0.00%
20	1.8s (1×)	100%	6.54s (3.6×)	90%	–	0%	0.27%
30	8.2s (1×)	97%	369.39s (45.1×)	32%	–	0%	0.06%
40	36.64s (1×)	80%	–	0%	–	0%	0.70%

Table 3.2: Computational results for the purely random instances in the harmonic case (basic PMP).

	IP-formulations						Heuristic
# tasks	BF		CF		TIF		FF
10	0.01s	100%	0.35s (33.1×)	100%	2.79(265.5×)	98%	0.00%
20	0.19s	99%	32.51s (174×)	66%	260.3(1393×)	50%	1.26%
30	0.45s	99%	487s (1072×)	3%	–	0%	0.76%
40	1.17s	98%	–	0%	–	0%	1.36%
50	2.96s	98%	–	0%	–	0%	0.63%
60	7.25s	97%	–	0%	–	0%	0.85%
70	12.76s	95%	–	0%	–	0%	0.00%
80	28.47s	94%	–	0%	–	0%	0.09%
90	45.58s	89%	–	0%	–	0%	0.00%
100	113.89s	90%	–	0%	–	0%	0.00%
150	977.97s	74%	–	0%	–	0%	0.00%

Table 3.3: Computational results for the RWP instances (harmonic case) for the basic PMP.

formulation.

Discussion. In the three settings of the harmonic case, the bin-formulation clearly outperforms the two other IP-formulations. While for small instances the congruence formulation is still competitive, as the number of tasks increases, the bin formulation becomes superior. The time-indexed formulation failed to find an optimal solution before the timeout even on small instances with ten tasks. This shows that taking the bin structure into account in the bin-formulation allows a significantly better running time in comparison with the other formulations. In contrast to the congruence formulation, integer variables are always "binary". As state of the art solvers cannot handle non-binary integer variables well, this is one advantage of the bin formulation. Also, the number of variables is a lot smaller than in the time-indexed formulation.

The First-Fit heuristic performs very well for the basic PMP and finds an optimal solution for most instances. Even though theoretically First-Fit is only a 2-approximation algorithm, it performs much better in practice. However, for real-world data we need to consider the extended PMP. In these instances First-Fit mostly missed the optimum by a significant margin. Also, it cannot provide a certificate of optimality and hence, in real settings one has to resort

	IP-formulations						Heuristic
# tasks	BF		CF		TIF		FF
10	0.09s	100%	0.2s (2.4×)	100%	5.03(58.7×)	100%	4.02%
20	2.16s	99%	19.96s (9.2×)	85%	29.19(13.5×)	15%	15.15%
30	19.8s	99%	119.22s (6×)	20%	–	0%	27.81%
40	97.02s	93%	–	0%	–	0%	25.13%
50	401.75s	62%	–	0%	–	0%	28.40%
60	655.06s	30%	–	0%	–	0%	14.12%
70	644.54s	8%	–	0%	–	0%	33.33%

Table 3.4: Computational results for the RWP instances (harmonic case) for the extended PMP.

to IP-formulations. Nevertheless, First-Fit can be used as a fast heuristic which computes an upper bound on the number of needed machines.

Real-World Instances

We solved each real-world instance from our industrial partner in less than 15 minutes to optimality. The most challenging one consists of 177 tasks, and an optimal solution uses 16 machines. The arising period lengths were 50, 100, 200, 400, 800, 1000, and 2000. Note that this instance is not harmonic. Nonetheless, the number of tasks having one of the problematic period lengths (that is, 1000 and 2000) were very small (three and six respectively). We transform the instance to be harmonic by taking the 3 tasks with period length 1000 and changing their periods to 200, and changing the 6 tasks with period 2000 to have period length 400. Note that a solution of the modified instance can easily be converted to a solution of the original instance. On the other hand, we could prove that the optimal solution of the restrictive instance is also optimal for the original instance since the separation constraints already contained a set of 16 tasks that had to be assigned to different machines.

4 Resource Constrained Scheduling: Computing Schedules Offline

4.1 Introduction

Modern computer architectures allow for a high degree of parallelization: Multi-core CPUs and GPUs provide dozens (in the case of GPUs even hundreds) of parallel processing units. However, these processing units cannot operate fully independent of each other. An example for such a shared resource is the cache of a multi-core processor. Now suppose that we are given a set of jobs that should be executed on these processing units. We would like all of these jobs to be finished as soon as possible, which requires to make efficient use of our given system. In other words, we would like to compute a *schedule* that assigns to each job a time interval of sufficient length on one of the processing units for exclusive usage in such a way that the time where the last job finishes is minimized. The fact that the processing units share a common resource adds additional complexity to the problem. When active, a job requires a certain part of the resource. The scheduler must ensure that the total resource requirement of jobs that are active simultaneously never exceeds the given limit.

In this chapter we characterize the algorithmic complexity of this problem similar to what we did for the periodic maintenance problem in Chapter 2. We study approximation algorithms and inapproximability results for several variants of the problem to get an overview over its complexity landscape. Unlike the periodic maintenance problem, there already exists a substantial amount of previous work in that direction. We give an overview, review some of the results in more detail and improve on others.

The problem has versatile applications, not only to model shared memory on parallel processors as previously mentioned. For example it could also be used to assign a limited amount of workers to different maintenance tasks, or to cope with the limited energy available for computing tasks. Next to the practical applications, the problem is also quite intriguing from a theoretical perspective. It can be seen as a hybrid of two classical problems from two different domains. On the one hand, one obtains classical minimum makespan scheduling problems if all resource requirements are zero. On the other hand, the well known BIN-PACKING problem is a special case of the resource constrained scheduling problem. The hybrid nature of the

problem is also reflected in our algorithms. To tackle the problem, we combine ideas and techniques from both domains.

We assume that all jobs are known ”in advance“ and we seek to find a good schedule before the system is actually running. This is known as *offline scheduling*. A fundamentally different concept is *online scheduling*. Here the jobs are not known in advance, but we rather learn about them over time, when execution of some jobs has already started. Initially only a subset of jobs is known, and we can compute a schedule for them. If at some point in time t, we learn about a new job, we are allowed to recompute the schedule. However we are not allowed to alter the schedule for the time before t. Online scheduling requires fundamentally different ways of treatment which we discuss in Chapter 5.

4.1.1 The Model and Basic Notation

We now define the problem more formally. In the *resource constrained scheduling problem*, one is given a set $\mathscr{J}$ of *n jobs* and a set of *m machines*. Every job j comes with a processing time $p(j)$ and a *resource requirement* $r(j)$. A schedule assigns to each job a time interval of length (at least) $p(j)$ on one of the m machines for exclusive usage. The *makespan* of the schedule is the largest endpoint of one of the intervals, i.e., the finishing time of the last job. The resource requirement specifies the amount of the common shared resource the job requires while it is active (i.e., during the time-interval it got assigned by the schedule) in order to work properly. For a time-index t, the *resource usage* at t is the sum of resource requirements of jobs active at time t. A schedule is *feasible* if the resource usage never exceeds a given limit. We always assume that this limit is 1. Note that this is without loss of generality this can always be ensured by scaling all resource requirements uniformly. The goal is to compute a feasible schedule of minimum makespan. For notational convenience, we usually denote an instance for the problem just by the job set $\mathscr{J}$, implicitly assuming that m, p and r are given as well. For a set of jobs $J \subseteq \mathscr{J}$, we use the shortcut $p(J) := \sum_{j \in J} p(j)$ for its total processing time.

This is the standard model that we will be concerned with in most of this chapter. However there exist several variations and generalizations that have been studied as well. One variation is regarding the type of schedule. The way it is defined above, each job has to be executed in one single run without interruptions. These kind of schedules are called *non-preemptive*. In contrast, *preemptive* schedules allow that jobs can be interrupted: their execution can be fragmented as long as these fragments sum up to the total processing time. If parts of the same job are even allowed to run on different machines, one says that the schedule allows *migration*. We stress that it is not allowed though that several parts of the same job are executed in parallel. A generalization of the model is regarding the machines. So far we assumed that all machines are *identical*, i.e., processing time and resource requirement of a job is independent of the machine it is assigned to. In contrast, for *unrelated machines*, the processing time and resource requirement of a job depends both on the job and the machine. There is no relation whatsoever between processing time/resource requirement of the same job on two

different machines. Another generalization is regarding the number of resources. Instead of one kind of resource, one can consider several independent ones. Each job then has a resource requirement for each of the resource types, and the scheduler must ensure that for none of the resource types, the sum of the respective resource requirements of simultaneously active jobs exceeds the limit of 1. Unless stated otherwise, when talking about the resource constrained scheduling problem, we always refer to the standard model (identical machines, no preemption, one resource).

4.1.2 Related Work

To give an extensive overview over the history of the problem and also to see how the computational complexity of the resource constrained scheduling problem relates to the basic problem of makespan minimization scheduling, we first discuss the latter before turning our attention to related work for the resource constrained scheduling problem.

The basic problem: Makespan minimization scheduling

When ignoring the resource constraint (or having all resource requirements set to zero), one obtains a classic combinatorial optimization problem, the *(makespan minimization) scheduling problem on identical machines*. As shorthand, from now on we will refer to this as *the basic problem*. It is a widely studied problem, and we only cite literature that is most relevant here. A seminal approximation algorithm for the problem is a greedy list scheduling algorithm by Graham [Gra66, Gra69]: On input $\mathscr{J}$, the algorithm constructs a schedule φ iteratively by adding the jobs one by one in some order: while there is an unscheduled job, find the earliest time where a machine turns idle and allocate an interval of suitable length for that job on this machine. Graham has shown that this simple algorithm is a 2-approximation algorithm. He also showed that if the jobs are added in non-increasing order of the processing times, one obtains a $\left(\frac{4}{3} - \frac{1}{3m}\right)$-approximation algorithm.

This is not the best approximation algorithm known in the literature. After a series of improvements, Hochbaum and Shmoys present a PTAS [HS87] for the problem. This is best one can hope for as the problem is NP-hard [GJ79] even in the strong sense. Already for instances where the number of machines is bounded by a constant, the problem is NP-hard (not in the strong sense though) [GJ79]. For this simpler case, there is an FPTAS by Sahni [Sah76].

Changing the machine model from identical to unrelated machines increases the computational complexity: Lenstra, Shmoys and Tardos [LST90] have shown that it is NP-hard to approximate the problem on unrelated machines with a factor better than $\frac{3}{2}$. At the same time, they give a 2-approximation algorithm for this variant in [LST90]. As for identical machines, if the number of machines is upper bounded by a constant, there is an FPTAS by Horrowitz and Sahni [HS76].

Resource constrained scheduling

Also the resource constrained scheduling problem is quite a classic problem and there is a substantial amount of literature related to it. Again we restrict to work that is most relevant for our purposes. The problem was first studied by Garey and Graham [GG75], even in the more general variant of having several types of resources. For s different resource types, they show that greedy list schedulers obtain an approximation ratio of $s+2-\frac{2s+1}{m}$. For the single resource case this ratio is $3-\frac{3}{m}$. The algorithm is explained in Section 4.2.1 as it is used as a subcomponent in our new algorithms. Garey and Johnson showed that the problem is NP-hard in the strong sense, even if the number of machines is bounded by a constant [GJ75].

Jansen and Porkolab studied the variant where both preemption and migration is allowed, also in the setting with several resources. They show that if s, the number of resource types, is bounded by a constant, then there is a polynomial time approximation scheme for the problem [JP06]. If the number of machines is upper bounded by a constant, the problem can even be solved optimally in polynomial time [BLK83].

Coming back to non-preemptive schedules, but altering the machine model to unrelated machines, there is related work by Grigoriev, Sviridenko and Uetz. The problem they consider is a generalization of the resource constrained scheduling problem (with one resource). Here the scheduler is allowed to determine the resource requirement of each job. Its running time then depends on the amount of resource assigned. They give 3.75-approximation algorithm for the setting of unrelated machines [GSU07]. Keller has studied their problem on identical machines and gives a $(3.5+\epsilon)$-approximation scheme [Kel08].

There are classifications for a multitude of further variations and restrictions in NP-hard or polynomial time solvable problems. We refer to [GJ75] and in particular [BLK83] for an extensive overview. Finally, there is another generalization of the resource constrained scheduling problem that is a very active subject of study: the *Resource Constrained Project Scheduling*. This problem however is defined in such generality that a theoretical study of this problem class is next to impossible. More practice oriented research branches however heavily work on this problem class. As computational studies are not or focus here, we refer to [ADN10, HB10] for surveys on this problem.

4.1.3 Our Contribution

Our new results are concerning the non-preemptive resource constrained scheduling problem on identical machines, i.e., the standard model. Our main result is a $(2+\epsilon)$-approximation scheme. This improves on the approximation bound of 3 by Garey and Grahams list scheduler [GG75]. If m, the number of machines, is upper bounded by some constant, this can even be improved to a PTAS. We also obtain a PTAS if we have no restriction on the number of machines, but require that the number of *different* resource requirements is upper bounded by a constant.

We also show that the problem is hard to approximate with a factor better than $\frac{3}{2}$. The reduction is pretty straightforward though, for that reason it is surprising that it was not published before. However to the best knowledge of the author, this is not the case. This also shows that our $(2+\epsilon)$-approximation narrows the gap between inapproximability and best approximation factor significantly.

Regarding the unrelated machine model, as mentioned previously, Grigoriev, Sviridenko and Uetz [GSU07] have shown that there is a 3.75-approximation algorithm for an even more general problem. In independent work, we have shown that there is a 4-approximation algorithm for the resource constrained scheduling problem on unrelated machines. Although being outperformed by [GSU07], we present our independent 4-approximation algorithm here anyway. The ideas for both algorithms are similar, but as we deal with a special case of the problem, our algorithm and its analysis is less technical. We also discuss how to transfer an insight from [GSU07] to turn our algorithm into a 3.75-approximation, achieving the same approximation ratio.

The complexity landscape

Similar to the theoretical analysis of the periodic maintenance problem in Chapter 2, we are interested in a rigorous account on the complexity landscape of the variants of the resource constrained scheduling problem by assessing lower and upper bounds on the best approximation ratio. As it might have become apparent to the reader in the discussion of related work, there are a lot of variants of the problem, and many of them have been studied intensively. In order to keep the presentation simple, we restrict to three dimensions of variants. We differentiate between the identical and unrelated machine model, consider preemptive and non-preemptive schedules and distinguish between the case where the number of machines is upper bounded by a constant or there are no restrictions on the number of machines. So in total we consider 8 different variants of the resource constrained scheduling problem. We list the known results for these variants in Table 4.1. For each variant, the table lists the best inapproximability and best approximation factors that where known before or that we found. Our new results are printed in bold face. We stress that for the preemptive cases, we allow migration in the case of identical machines, but do not allow it for unrelated machines. The reason is that this better reflects the properties of the real-world systems. For identical machines like multi-core processors, allowing migration is usually not more “costly” than allowing preemption. This is usually different on heterogeneous hardware that is modeled by unrelated machines.

Comparing resource constrained scheduling with basic problem

We now briefly discuss on how the computational complexity changes by adding the resource constraint. In the case of identical machines with preemption and migration, the basic problem can be solved easily with a greedy type algorithm. Here we observe an increase in

	Constant no of machines	Arbitrary no of machines
Identical parallel machines		
Preemp./Migr.	Polytime solvable [BLK83]	NP-hard, PTAS [JP06]
Non-preemptive	NP-hard [GJ75], **PTAS**	$(\frac{3}{2}-\epsilon)$**-hard to approximate** $(2+\epsilon)$**-approximation algorithm**
Unrelated parallel machines		
Preemptive	NP-hard [GJ79]	$(\frac{3}{2}-\epsilon)$-hard to approximate [LST90] 3.75-approximation algorithm [GSU07]
Non-preemptive	NP-hard [GJ75]	$(\frac{3}{2}-\epsilon)$-hard to approximate [LST90] 3.75-approximation algorithm [GSU07]

Table 4.1: The complexity landscape of the resource constrained scheduling problem.

complexity as the resource constrained scheduling problem is NP-hard. In the non-preemptive version, an FPTAS [Sah76] is known for the basic setting if the number of machines is constant, and a PTAS [HS87] is known for the unbounded case. For both variants, the complexity increases with the introduction of the resource constraint, as now the constant-machine case is already strongly NP-hard, and the unbounded machine variant is APX-hard.

For the machine model of unrelated machines, there are no new inapproximability bounds for the resource constrained scheduling problem. They come from the basic setting [LST90]. However, while there is are 2-approximation algorithm for the basic setting, only a 3.75-approximation algorithm is known for the resource constrained setting. In the case of constant number of machines, the difference is more drastic: While there is an FPTAS [HS76] for the basic setting, we only have the 3.75-approximation for the resource constrained scheduling problem from the unbounded machine setting. The computational complexity increases here as the resource constrained scheduling problem is strongly NP-hard on a constant number of machines, because this is true already on identical machines.

4.1.4 Outline

The organization of this chapter is as follows. We first deal with the (standard) machine model of identical machines (Section 4.2). After giving a formal definition of the resource constrained scheduling problem and introducing some basic notation used throughout the chapter, in Section 4.2.1 we review Garey and Grahams classical list scheduler algorithm which achieves an approximation guarantee of 3. This can be seen as a warm-up for the highlight of this chapter, the $(2+\epsilon)$-approximation scheme which we present in Section 4.2.2. It is followed by a short discussion of (rather straightforward) reductions that lead to the claimed inapproximability bounds (Section 4.2.5). We then turn our attention to the machine model of unrelated machine and present the 3.75-approximation algorithm in Section 4.3. We conclude the section with a discussion on generalizations and further research questions in Section 4.4.

4.2 Scheduling on Identical Machines

Before discussing the identical machines case, we introduce some more notation to formally describe schedules. A *slot* is a time-interval $[t_1, t_2)$ with $t_2 \geq t_1$. With $[m] := \{0, 1, \ldots, m-1\}$, a *machine slot* $(k, t_1, t_2) \in [m] \times \mathbb{Q}_{\geq 0} \times \mathbb{Q}_{\geq 0} := \mathscr{S}$ is a slot assigned to a machine k. We call $\mathscr{S}$ the set of all machine slots. For notational convenience we often identify machine slots s with the interval they represent: We write $|s|$ instead of $|[t_1, t_2)|$, and $t \in s$ instead of $t \in [t_1, t_2)$. The same applies for unions and intersections of machine slots.

A *schedule* is a map $\varphi : \mathscr{J} \to \mathscr{S}$ that assigns a machine slot to each job $j \in \mathscr{J}$. We call a schedule *feasible*, if the length of the machine slot assigned to each job is sufficient, assigned machine slots on the same machine are pairwise non-intersecting, and the resource constraint is satisfied. Formally, a schedule is feasible if $|\varphi(j)| \geq p(j)$ for all $j \in \mathscr{J}$, we have that $\varphi(j) \cap \varphi(j') = \emptyset$ for all $j, j' \in \mathscr{J}$ assigned to the same machine, and $\sum_{j \in \mathscr{J}:\, t \in \varphi(j)} r(j) \leq 1$ for all $t \in \mathbb{Q}_{\geq 0}$. If not noted otherwise, when talking about schedules we always mean feasible schedules. With our definition of schedules, the makespan is then $T(\varphi) := \sup \bigcup_{j \in \mathscr{J}} \varphi(j)$. As before, with $OPT(\mathscr{J})$ we denote the optimal makespan of an instance $\mathscr{J}$.

This definition of schedules via machine slots might appear unnecessarily complicated to the reader at first sight. However, it will turn out to be quite natural when discussing the main result, the $(2+\epsilon)$-approximation scheme for resource constrained scheduling on identical machines. For the sake of consistent notation, we use it throughout the whole section.

4.2.1 List Scheduler: A 3-Approximation Algorithm

Already in 1975, Garey and Graham [GG75] presented the following simple list scheduling algorithm and proved that it achieves an approximation ratio of 3. On input $\mathscr{J}$, the algorithm iteratively computes a schedule by adding the jobs one by one as follows. While there are unassigned jobs, determine the smallest time t a yet unassigned job could be placed into the schedule without making it infeasible. If there are several candidate jobs, pick a job according to some priority ordering. Assign the job by allocating a suitable machine slot $(k, t, t + p(j))$ on an idle machine k. A more formal description of the algorithm can be found in Listing 2. It is easy to see that this algorithm has polynomial running time and that it produces a feasible schedule. Garey and Graham [GG75] have shown that it is a 3-approximation algorithm, no matter which priority ordering is chosen. To be more precise, the statement from their paper actually is weaker: They state that for any instance of the problem, the schedule generated by the list schedule using the worst ordering is at most 3 times longer than the one generated using the best ordering. Unlike the basic setting without the resource constraint, this does *not* automatically imply that it is a 3-approximation algorithm: unlike the basic setting, there are instances where no ordering produces a schedule with optimal makespan. The reason is that there are instances where every optimal solution requires to introduce some idle times on some processors despite the fact that jobs of suitable resource requirement are waiting. These idle times cannot be enforced in schedules generated by list schedulers. However, reviewing

Algorithm 2 A list scheduler for resource constrained scheduling

```
ℒ ← (j_1, j_2, ..., j_n) {Jobs given in some order}
t ← 0
while ℒ has unscheduled jobs do
  for i = 1 to n do
    if j_i is yet unscheduled and r(j_i) does not lift resource usage above 1 then
      if ∃ machine k idle at time t then
        Allocate machine slot (k, t, t + p(j_i)) and assign it to job j_i.
      end if
    end if
  end for
  t ← next finishing time of a job.
end while
```

the proof of Gary and Graham the stronger statement easily follows.

Although the approximation quality of this algorithm is dominated by the $(2+\epsilon)$-approximation scheme, it is still interesting for us as it will be used as a sub-component in the approximation scheme. Next we will prove that, when using the resource requirements to determine the priority ordering, the list scheduler is a 3-approximation algorithm. This is a slightly weaker statement than what follows from Garey and Grahams proof (which works for arbitrary priority orderings), but the proof is much simpler. Along the way we show better tailored approximation guarantees for special kinds of instances, which will be very useful for the $(2+\epsilon)$-approximation scheme later. To this end, consider a schedule φ computed by the list scheduler for an instance $\mathscr{J}$, when using the resource requirements as priority ordering, jobs of largest resource requirement having best priority. Let j^* be a job that finishes last, i.e., a job that determines the makespan. We first discuss the (easy) case that j^* is a job using more than half of the resource.

Lemma 4.2.1. *If $r(j^*) > \frac{1}{2}$, then the schedule is optimal, i.e., $T(\varphi) = OPT(\mathscr{J})$.*

Proof. Let $j_1, \ldots, j_k$ be the jobs with $r(j_i) > \frac{1}{2}$ (sorted by resource requirement). Observe that the algorithm schedules these jobs sequentially in that order: At time 0, job j_1 is scheduled as it has the largest resource requirement. Whenever a job j_i finishes, there is enough resource available for job j_{i+1}. Also, at that time, j_{i+1} is the job of highest resource requirement not yet scheduled and is hence chosen to be scheduled next.

We conclude that $j^* = j_k$. As the optimal schedule also has to schedule the jobs $j_1, \ldots, j_k$ sequentially, we conclude that our schedule is optimal. □

We now deal with the case that $r(j^*) \leq \frac{1}{2}$. The analysis of the algorithms performance guarantee is based on two simple lower bounds on the makespan of an optimal solution (which already where introduced by Garey and Graham). One bound is given by the total processing

time of the instance: Per time unit, every schedule can "process" at most m units of processing time by keeping all machines busy. Hence, to process all jobs completely, every schedule needs at least $\sum_{j\in\mathscr{J}} p(j)/m$ time-units. For the second bound, define the *resource consumption* of a job j as the resource/processing product $r(j)\cdot p(j)$. As the resource usage of every schedule at all times is at most 1, a schedule can "process" only one unit of resource consumption per time-unit. Consequentially, every schedule needs at least $\sum_{j\in\mathscr{J}} r(j)\cdot p(j)$ time-units to process all the resource consumption of the instance. We conclude:

Lemma 4.2.2. *We have the following lower bounds for the optimal makespan:*

$$OPT(\mathscr{J}) \geq \bar{P}(\mathscr{J}) := \sum_{j\in\mathscr{J}} p(j)/m, \qquad OPT(\mathscr{J}) \geq \bar{R}(\mathscr{J}) := \sum_{j\in\mathscr{J}} p(j)\cdot r(j)$$

We now analyze the structure of the computed schedule. It is easy to see that for each time-index $t < T(\varphi)$, we are in at least one of the following three cases:

(a) all machines are busy,

(b) not all machines are busy and job j^* is *not* active,

(c) job j^* is active.

Let $\beta := \min\left\{\frac{1}{2}, \max_{j\in\mathscr{J}} r(j)\right\}$.

Lemma 4.2.3. *The total time spent in case (a) is at most $\bar{P}(\mathscr{J})$. The total time spent in case (b) is at most $\frac{1}{1-\beta}\bar{R}(\mathscr{J})$*

Proof. If all machines are busy for $\bar{P}(\mathscr{J})$ time-units, then the total amount of processing done is $m\cdot\bar{P}(\mathscr{J}) = p(\mathscr{J})$. Hence all jobs are finished afterwards. We conclude that all machines can be busy simultaneously only for at most $\bar{P}(\mathscr{J})$ time units and the bound for case (a) follows.

To deal with case (b), we show that for each time-index t spent in case (b), the resource usage at that time exceeds $1-\beta$. First assume that $r(j^*) \leq \beta$. Now assume that at time t in case (b), the resource load is at most $1-\beta$. Then job j^* (or some job with higher resource requirement) could and would have been scheduled to one of the available machines, thus ending up in case (a) or (c). Hence if $r(j^*) \leq \beta$, then the resource load is always more than $1-\beta$. On the other hand, if $r(j^*) > \beta$, then by definition we have $\beta = \frac{1}{2}$ and hence $r(j^*) > \frac{1}{2}$. As seen in the proof of Lemma 4.2.1, this implies that there is always a job of resource requirement of more than $\frac{1}{2}$ running throughout the schedule, and hence the resource usage is more than $\frac{1}{2} = 1-\beta$.

Now we know that during case (b), the resource usage is always at least $1-\beta$. Let t_1 be the total time spent in case (b). We conclude that the total resource consumption "processed" during case (b) is at least $t_1\cdot(1-\beta)$. On the other hand, the schedule does not process more resource consumption than $\bar{R}(\mathscr{J})$ and the bound for case (b) follows. □

This already implies that the algorithm is at least a 4-approximation. Depending on the parameters of the instance, the approximation guarantee can be much better, which will actually be exploited when the algorithm is used as a sub-procedure for the $(2+\epsilon)$-approximation algorithm.

Lemma 4.2.4. *Algorithm 2 is a* 4*-approximation algorithm. Let* $\tau := \max_{j\in\mathscr{J}} p(j)$ *and* $\beta := \min\left\{\frac{1}{2}, \max_{j\in\mathscr{J}} r(j)\right\}$*. More precisely, the makespan of the schedule computed by the algorithm is at most*

$$\bar{P}(\mathscr{J}) + \frac{1}{1-\beta}\bar{R}(\mathscr{J}) + \tau.$$

Proof. With Lemma 4.2.3, we get that the total time spent in cases (a) and (b) together is at most $\bar{P}(\mathscr{J}) + \frac{1}{1-\beta}\bar{R}(\mathscr{J})$. The total time of case (c) is $p(j^*) \le \tau$. This implies the approximation ratio of 4 using the bounds from Lemma 4.2.2, together with the fact that $\tau \le OPT$ and $\beta \le \frac{1}{2}$. □

With a more careful analysis, one can also show that the algorithm is a 3-approximation algorithm. This will not be relevant for the $(2+\epsilon)$-approximation algorithm. However we present the proof for the sake of completeness.

Lemma 4.2.5. *The time spent in cases* (b) *and* (c) *together is at most* $2\cdot OPT$*.*

Proof. If $r(j^*) > \frac{1}{2}$, the statement is trivially true as by Lemma 4.2.1, the length of the total schedule is OPT. So we assume that $r(j^*) \le \frac{1}{2}$. Consider a time-index t of case (b). The total resource load at time t must be more than $1 - r(j^*)$, as otherwise j^* (or some job with higher resource requirement) could and would have been scheduled to one of the available machines, thus ending up in case (a) or (c).

Let t_1 (t_2) be the total time spent in case (b) (case (c) respectively). Clearly $t_2 = p(j^*) \le OPT$. Let $\ell \in [0,1]$ be so that $t_2 = \ell \cdot OPT$. Then the total resource consumption of case (b) and (c) together is at least $t_1 \cdot (1 - r(j^*)) + t_2 \cdot r(j^*)$. With Lemma 4.2.2 we conclude that

$$OPT \ge \bar{R}(\mathscr{J}) \ge t_1 \cdot (1 - r(j^*)) + t_2 \cdot r(j^*) = t_1 \cdot (1 - r(j^*)) + \ell \cdot r(j^*) \cdot OPT.$$

We rewrite this to

$$(1 - \ell \cdot r(j^*)) \cdot OPT \ge t_1 \cdot (1 - r(j^*))$$

and conclude

$$t_1 \le \frac{1 - \ell \cdot r(j^*)}{1 - r(j^*)} OPT = \left(1 + (1-\ell)\frac{r(j^*)}{1 - r(j^*)}\right) \cdot OPT \le (2 - \ell) \cdot OPT,$$

the last inequality being due to the fact that $r(j^*) \le \frac{1}{2}$. Finally

$$t_1 + t_2 \le (2 - \ell) \cdot OPT + \ell \cdot OPT = 2 \cdot OPT.$$

□

With Lemma 4.2.3 and Lemma 4.2.5 we conclude that the algorithm is a 3-approximation algorithm.

Theorem 4.2.6. *Algorithm 2 is a* 3*-approximation algorithm.*

A generalization

We will use this algorithm also in a small adaptation: Instead of computing a schedule from scratch, we start with a feasible schedule φ' for a subset of $\mathscr{J}' \subseteq \mathscr{J}$ of jobs of the instance. The algorithm gets this schedule as input and is asked to complete it by adding the remaining jobs of the instance. The algorithm works exactly the same way as before: As long as not all job are scheduled, the earliest time t where *some* job can be scheduled is computed. Then the algorithm schedules the job of largest resource requirement that is possible. The implementation details differ slightly from Listing 2, but it is straightforward to do the necessary adaptations to obtain a polynomial time algorithm.

An *activation point* of a schedule φ is a time-index t where some machine becomes busy that was idle before, or the resource usage increases. Let $A(\varphi')$ denote the number of activation points of the partial schedule φ'. Also let $\tau := \max_{j\in\mathscr{J}\setminus\mathscr{J}'} p(j)$ be the largest processing time of a job not scheduled by φ'. Moreover let $\beta := \max_{j\in\mathscr{J}\setminus\mathscr{J}'} r(j)$ be the largest resource requirement. We can bound the makespan of schedules generated by the algorithm as follows:

Lemma 4.2.7. *Let $\mathscr{J}$, $\mathscr{J}'$, φ', τ and β be as defined above. In polynomial time we can compute a schedule for $\mathscr{J}$ with $T(\varphi) \le \max\left\{T(\varphi'), \bar{P}(\mathscr{J}) + \frac{1}{1-\beta}\bar{R}(\mathscr{J}) + (A(\varphi')+1)\cdot\tau\right\}$.*

Proof. If $T(\varphi) = T(\varphi')$, then the statement is obviously true. Otherwise, we again bound the makespan of the total schedule by bounding the makespans of certain sub-parts separately. To this end, we introduce a forth case:

(d) φ' has an activation point in the interval $[t, t+\tau)$.

Observe that we are always in one of the cases (a) - (d). Clearly the total time spent in case (d) is at most $A(\varphi')\cdot\tau$. If we are *not* in case (d), note that the algorithm behaves exactly as the original variant (that started with an empty schedule). Hence, by the lines of the proofs of Lemma 4.2.3 and Lemma 4.2.4, we can show that the time spent in case (a) - (c) and *not* (d) is bounded by

$$\bar{P}(\mathscr{J}) + \frac{1}{1-\beta}\bar{R}(\mathscr{J}) + \tau.$$

Hence in total we get the claimed bound. □

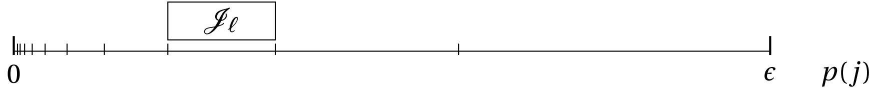

Figure 4.1: An illustration of the proof of Lemma 4.2.9.

4.2.2 The 2-Approximation Scheme

We now discuss the main result of this chapter, a family of $(2+\epsilon)$-approximation algorithms for the standard model of resource constrained scheduling: for each constant $\epsilon > 0$ there is a polynomial-time algorithm with an approximation-guarantee of $2+\epsilon$. To this end, fix a constant $\epsilon > 0$. For the sake of non-ambiguous notation, if we write $O(1)$, we mean a constant that is *independent* on the constant ϵ. We write $O_\epsilon(1)$ for a constant that does depend on ϵ. The second form of notation is useful to hide the exact dependence on ϵ where it is not needed, e.g., in running times of sub components of algorithms. The first form of notation is used when the exact dependence of the constant on ϵ is important, e.g., when computing the makespan of certain schedules.

We actually present a polynomial time algorithm with the following property. For any instance $\mathscr{J}$ with $OPT(\mathscr{J}) \leq 1$, it computes a feasible schedule of makespan at most $2+O(1)\cdot\epsilon$. Such an algorithm can then easily be turned into a $(2+\epsilon)$-approximation algorithm using standard techniques like a binary search framework. To not distract from the more relevant ideas of the algorithm, these standard techniques are discussed in more detail at the end of the presentation of the algorithm in Section 4.2.4. We simplify the problem even further by considering only instances that are (γ, M)-restricted.

Definition 4.2.8. *Let* $1 \geq \gamma > 0$ *and* $M > 1$. *An instance* $\mathscr{J}$ *is* (γ, M)-restricted *if for each job* j *it holds that either* $p(j) \geq \gamma$ *or* $p(j) < \gamma/M$.

In (γ, M)-restricted instances, we classify jobs j with $p(j) \geq \gamma$ as *large* and jobs j with $p(j) < \gamma/M$ as *tiny*. Every large job is at least M times as long as any tiny job. This fact will be crucial for our algorithm later. For a (γ, M)-restricted instance $\mathscr{J}$, let $\mathscr{J}_{\text{large}}$ be the set of large and $\mathscr{J}_{\text{tiny}}$ be the set of tiny jobs respectively. The following lemma justifies to reduce to restricted instances as it allows us to make every instance restricted by removing a set of jobs that can be scheduled separately using negligible makespan.

Lemma 4.2.9. *For any choice of constants* $\epsilon > 0$ *and* $M > 1$, *there is another constant* $\gamma^*_{\epsilon,M}$ *such that for any instance* $\mathscr{J}$, *in polynomial time we can find a value* $\gamma \geq \gamma^*_{\epsilon,M}$ *and a partition of the instance* $\mathscr{J} = \mathscr{J}_1 \dot\cup \mathscr{J}_2$ *so that* $\mathscr{J}_1$ *is* (γ, M)*-restricted and for* $\mathscr{J}_2$ *the list scheduler from Section 4.2.1 computes a schedule of makespan at most* $O(1)\cdot\epsilon\cdot OPT(\mathscr{J})$.

Proof. Let $K := \lceil 1/\epsilon \rceil$ be a constant. Consider the sequence of points $\epsilon \cdot M^{-\ell}$ for $\ell = 1, \ldots, K$ as illustrated in Figure 4.1. These points define K many disjoint (half open) intervals with the property that the right endpoint is M times larger than the left endpoint. Let $\mathscr{J}_\ell$ be the set of

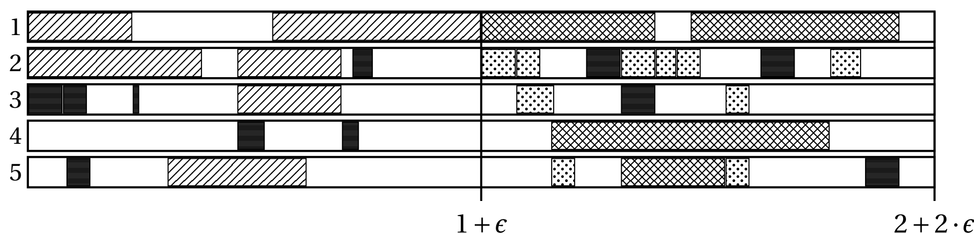

Figure 4.2: A conceptual schedule computed by the $(2+\epsilon)$-approximation algorithm. The striped jobs are the ones assigned in step 1a. The hatched ones are those assigned in step 1b. The dotted ones are assigned in step 2a, and the dark ones are added in step 2b.

jobs whose processing times are contained in the ℓth interval. By some extended pigeonhole principle we can show that for one of the K intervals, say the ℓth, both resource consumption and processing load are low, i.e., $\bar{R}(\mathscr{J}_\ell) \le \frac{2}{K}\bar{R}(\mathscr{J})$ and $\bar{P}(\mathscr{J}_\ell) \le \frac{2}{K}\bar{P}(\mathscr{J})$. (Recall the definitions of $\bar{P}$ and $\bar{R}$ from Lemma 4.2.2.) The argument is as follows: As the intervals are disjoint, for at least $K/2$ of them, their resource consumption cannot exceed $\frac{2}{K}\bar{R}(\mathscr{J})$, as otherwise their resource consumption would sum up to more than the total resource consumption of the instance. Amongst those $K/2$ many intervals, by the pigeonhole principle there must be one that has processing load at most $\frac{2}{K}\bar{P}(\mathscr{J})$.

For this interval ℓ, observe that by construction also $p(j) \le \epsilon$ for all $j \in \mathscr{J}_\ell$. Hence Lemma 4.2.4 tells us that the list scheduler invoked on instance $\mathscr{J}_\ell$ computes a schedule of makespan at most $O(1) \cdot \epsilon \cdot OPT(\mathscr{J})$. Hence setting $\mathscr{J}_2 := \mathscr{J}_\ell$, $\mathscr{J}_1 := \mathscr{J} \setminus \mathscr{J}_2$ and γ to the right endpoint of $\mathscr{J}_\ell$ given the desired partitioning. Note that $\gamma \ge \gamma^*_{\epsilon,M} := \epsilon \cdot M^{-\lceil 1/\epsilon \rceil}$ and $\gamma^*_{\epsilon,M}$ is constant. □

We set $M := \epsilon^{-3}$. For the remainder of this section, let $\mathscr{J}$ be a (γ, M)-restricted instance with $OPT(\mathscr{J}) \le 1$. The algorithm to schedule restricted instances is in itself composed of several steps. The rough outline is as follows. We first consider only the large jobs. We compute a schedule of makespan $1+\epsilon$ for all except a constant number of those jobs. Next we compute a set of candidate schedules for the remaining large jobs. Each of them has makespan $1+\epsilon$ as well. We can guarantee that for at least one of the schedules, all tiny jobs can be added without increasing the makespan (we do not know which one though, so we try them all). As the problem of adding the tiny jobs is NP-hard on its own, we again rely on approximations by allowing the makespan to increase "a bit". Adding tiny jobs takes place in two steps. First, for each candidate schedule we add the *fat* jobs, i.e., tiny jobs of "large" resource requirement. This is successful at least for one candidate. We then concatenate the successful schedule with the separate schedule that we computed in the first step. Finally we complete the schedule by adding the so far not yet scheduled *thin* jobs, i.e., tiny jobs with "small" resource requirement, using the list scheduler from Section 4.2.1. In total this will result in a schedule of makespan $2 + O(1) \cdot \epsilon$. Figure 4.2 illustrates the schedule.

In summary, we have the following steps.

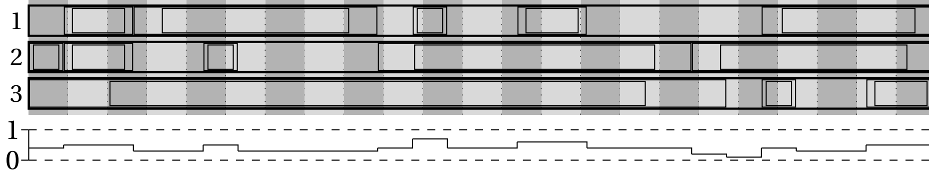

Figure 4.3: A feasible schedule scaled by a factor of $1+\epsilon$. The inner black boxes within the slots depict the processing time of the jobs assigned to the slots. The difference between slot-width and job-width illustrates the slack.

Step 1a: Compute schedule φ_1 for "almost" all large jobs.

Step 1b: Compute set of candidate schedules for remaining large jobs.

Step 2a: For each candidate, try to add tiny fat jobs.

Concatenate φ_1 with successful schedule from step 2a.

Step 2b: Add tiny thin jobs to concatenation using the list scheduler.

We proceed by discussing each step in more detail.

Step 1a: Scheduling "almost" all large jobs.

Scheduling large jobs is done using a combination of enumeration and linear programming techniques. To make the enumeration work, we discretize the scheduling decisions for large jobs as follows. We set $\delta := \epsilon \cdot \gamma/2$ and require that the machine slots assigned to large jobs start and end at integer multiples of δ. I.e., for each $j \in \mathcal{J}_{\text{large}}$, its machine slot should be of the form $(k, \ell_1 \cdot \delta, \ell_2 \cdot \delta)$ with $\ell_1, \ell_2 \in \mathbb{N}_0$. We call schedules whose large jobs have this property *δ-atomic*. Note that we do not pose any restrictions on non-large jobs in δ-atomic schedules. The following lemma justifies that it is sufficient to consider this type of schedules.

Lemma 4.2.10. *There exists a δ-atomic schedule for $\mathcal{J}$ whose makespan is at most* $(1+\epsilon) \cdot OPT(\mathcal{J}) \leq 1+\epsilon$.

Proof. Take an optimal schedule φ. Define another schedule φ' by setting $\varphi'(j) := (k, (1+\epsilon) \cdot a, (1+\epsilon) \cdot b)$ for each $\varphi(j) = (k, a, b)$. Observe that φ' is feasible: No slots assigned to the same machine overlap and the machine slots assigned to jobs are at least as large as in φ. Also the resource constraint is satisfied as any set of machine slots active at time t in φ' are active at time $(1+\epsilon)^{-1} \cdot t$ in schedule φ. By construction, the makespan is at most $(1+\epsilon) \cdot T(\varphi) = (1+\epsilon) \cdot OPT(\mathcal{J})$.

We now turn φ' into a δ-atomic schedule. See Figure 4.3 for an illustration of φ'. For every large job $j \in \mathcal{J}_{\text{large}}$, its machine slot $\varphi'(j)$ has length $|\varphi'(j)| = (1+\epsilon) \cdot |\varphi(j)| \geq p(j) + 2\delta$. The last inequality is by the fact that j is large and hence $\epsilon \cdot p(j) \geq \epsilon \cdot \gamma = 2\delta$. Hence, we can round the

starting times of large machine slots up and their ending times down to multiples of δ without decreasing their length below $p(j)$. This way we obtain a feasible δ-atomic schedule. □

Aiming for δ-atomic schedules, there is only a constant number of possible starting times for each job. Still, enumerating all possible starting times for all jobs would lead to an algorithm with exponential running time. Our way around this is to enumerate machines slots without job assignment (which can be done in polynomial time), and then assigning jobs via a linear program. We call a set of $|\mathscr{J}_{\text{large}}|$ many machine slots (without job assignment) a *template*. It is called *feasible* if it corresponds to a feasible schedule (i.e., there is a feasible schedule that uses the slots from the template). To compute a schedule for the large jobs, we will first find a feasible template, and later assign jobs to its slots. More precisely, we want to find the template corresponding to the schedule of makespan $1+\epsilon$ from Lemma 4.2.10. To do so, we partition the time-line $[0, 1+\epsilon)$ into *frames* $F_\ell := [\delta \cdot (\ell - 1), \delta \cdot \ell)$ for $\ell \in \mathbb{N}$ and set $\mathscr{F} := \left\{F_\ell \,:\, 1 \le \ell \le \left\lceil \frac{1+\epsilon}{\delta} \right\rceil \right\}$. Observe that all large machine slots in our δ-atomic schedule are unions of consecutive frames from $\mathscr{F}$. Note that since $\gamma \ge \gamma^*_{\epsilon,M}$ (and $\gamma^*_{\epsilon,M}$ is a constant), we have that $|\mathscr{F}| = \left\lceil \frac{1+\epsilon}{\delta} \right\rceil = \left\lceil \frac{1+\epsilon}{\epsilon \cdot \gamma/2} \right\rceil = O_\epsilon(1)$ is constant. This allows us to find a feasible template by enumeration.

Lemma 4.2.11. *In polynomial time we can compute a polynomial number of candidate templates. At least one of them is feasible.*

Proof. In a feasible δ-atomic schedule of makespan $1+\epsilon$ there are at most $|\mathscr{F}|$ many large jobs on a single machine. This is because the processing times exceed the frame length. For each job, start and end frames are chosen from $|\mathscr{F}|$ many possibilities. Hence, there are at most $Q := \binom{|\mathscr{F}|}{2}^{|\mathscr{F}|}$ feasible combinations of machine slots for one machine. Up to permutation of machines, we can describe every template by specifying how many machines use each of the Q possibilities. This results in at most $m^Q = m^{O_\epsilon(1)}$ many candidate templates. One of them must correspond to the δ-atomic schedule from Lemma 4.2.10 and is thus feasible. □

To compute the schedule, we repeat the following procedure for each of the candidate templates. It will be successful for any feasible template. Let $\mathscr{T}$ be a template. We use a linear program to compute an assignment of large jobs to machine slots from $\mathscr{T}$. Let $\mathscr{I}$ denote the set of all slots from $\mathscr{T}$ (i.e., the time intervals without the machine assignment). For each interval $I \in \mathscr{I}$, let μ_I denote the number of machines slots from the template $\mathscr{T}$ that use it.

We model the problem of assigning jobs to intervals with the following linear program.

$$\sum_{I \in \mathscr{I},\ |I| \geq p(j)} x_{j,I} = 1 \qquad \forall j \in \mathscr{J}_{\text{large}} \tag{4.1}$$

$$\sum_{j \in \mathscr{J}_{\text{large}}} x_{j,I} \leq \mu_I \qquad \forall I \in \mathscr{I} \tag{4.2}$$

$$\sum_{I \in \mathscr{I}, F \subseteq I} \sum_{j \in \mathscr{J}_{\text{large}}} r(j) x_{j,I} \leq 1 \qquad \forall F \in \mathscr{F} \tag{4.3}$$

$$x_{j,I} \geq 0 \qquad \forall j \in \mathscr{J}_{\text{large}}\ \forall I \in \mathscr{I}$$

The variables $x_{j,I}$ model the assignment of jobs to intervals, where $x_{j,I} = 1$ means that job i is assigned a machine slot with time interval I. The constraints of type (4.1) ensure that every job gets assigned a slot. The constraints (4.2) make sure that we do not use more slots of the respective types than are available in the template. Finally the constraints of type (4.3) ensure that during each frame, the resource constraint is satisfied. Of course we cannot hope for finding an integer solution for this LP in polynomial time. However we can find a solution that assigns most of the jobs integrally as we show next. A fundamental property of the set of feasible solutions of a linear program is the following. If there is a feasible solution, then there is a feasible solution whose number of nonzero variables is bounded by the number of constraints (ignoring non-negativity constraints). This solution is called a *basic solution* or *extreme point solution*. In this LP, the number of constraints is “low”, which implies that most jobs get assigned integrally as we show next. For later steps, we also need to keep track of the number of activation points this schedule has. Recall the definition of activation points from Section 4.2.1 as time-indexes where a machine turns busy from idle, or the resource usage increases.

Lemma 4.2.12. *Given a feasible template, in polynomial time we can compute a δ-atomic schedule of makespan $(1+\epsilon)$ that schedules all except for $|\mathscr{I}|+|\mathscr{F}|$ many large jobs. The schedule has at most $O(1) \cdot \frac{1}{\epsilon \cdot \gamma}$ activation points.*

Proof. Observe that the number of constraints (without the non-negativity constraints) is $k := |\mathscr{J}_{\text{large}}| + |\mathscr{I}| + |\mathscr{F}|$. Hence an extreme point solution of the above linear program has at most k many non-zero entries. Due to Constraints (4.1), every job $j \in \mathscr{J}_{\text{large}}$ has at least one non-zero variable. We conclude that at most $|\mathscr{I}| + |\mathscr{F}|$ jobs are fractionally assigned. Given the template, it is straightforward to turn the integral sub-assignment of the LP solution into a feasible δ-atomic schedule for the integrally assigned jobs. Since it is δ-atomic, there are at most $|\mathscr{F}|$ many activation points. □

Note that $|\mathscr{I}| = O_\epsilon(1)$. We have seen before that $|\mathscr{F}| = O_\epsilon(1)$ as well. Hence all except for a constant number of jobs are scheduled.

Step 1b: Scheduling the remaining large jobs.

If there are only constantly many large jobs to schedule, we can enumerate all possible δ-atomic schedules in polynomial time. It will be crucial later that one of these schedules is *extendable*:

Definition 4.2.13. *Let $\mathscr{J}$ be a set of jobs and let φ be a schedule for a subset $\mathscr{J}' \subseteq \mathscr{J}$. The schedule φ is* extendable for *$\mathscr{J}$ if the jobs $\mathscr{J} \setminus \mathscr{J}'$ can be added to φ without increasing the makespan.*

Lemma 4.2.14. *Let $\mathscr{J}'_{\text{large}} \subseteq \mathscr{J}$ be a set of large jobs with $|\mathscr{J}'_{\text{large}}| = O_\epsilon(1)$. In polynomial time we can compute a set of δ-atomic schedules of makespan $1+\epsilon$. Each of them is feasible for the sub-instance $\mathscr{J}'_{\text{large}}$. At least one of them is extendable for $\mathscr{J}$. The number of activation points of each schedule is at most $O(1) \cdot \frac{1}{\epsilon\gamma}$.*

Proof. Observe that it is sufficient to restrict ourselves to the first $\min\left\{m, \left|\mathscr{J}'_{\text{large}}\right|\right\} = O_\epsilon(1)$ machines. Now for each job, we can choose one of those machines, a start- and an end-frame from $\mathscr{F}$. Hence there are $O_\epsilon(1) \cdot \binom{|\mathscr{F}|}{2}$ many possibilities per job. As there are only constantly many jobs, in total there are only constantly many combinations to consider. Hence we can check all of them in polynomial time, keeping only feasible ones as candidates. One of them must be the sub-schedule for $\mathscr{J}'_{\text{large}}$ of the δ-atomic schedule from Lemma 4.2.10 (up to permutation of machines). We conclude that this schedule is extendable. Again, the bound on the number of activation points follows from the fact that the generated schedules are δ-atomic. □

Step 2a: Adding tiny fat jobs.

We now describe how to add tiny jobs to a δ-atomic schedule for large jobs. As mentioned previously, there are two kinds of tiny jobs that need to be treated differently. Define $\beta := \epsilon$. We say that a tiny job j is *fat* if $r(j) \geq \beta$. Otherwise it is *thin*. We are now in the following situation: We consider a sub-instance $\mathscr{J}' \subseteq \mathscr{J}$ consisting of some large jobs $\mathscr{J}'_{\text{large}}$ and all tiny fat jobs $\mathscr{J}'_{\text{tiny}}$. We stress that $\mathscr{J}'_{\text{tiny}}$ does *not* contain the tiny thin jobs. Given a schedule φ_2 for $\mathscr{J}'_{\text{large}}$ of makespan $1+\epsilon$, and we want to add the tiny fat jobs $\mathscr{J}'_{\text{tiny}}$. For simplicity we assume that φ_2 is extendable and show that in this case, the algorithm is successful. The road-map is as follows. We first round the resource requirements so that only a constant number of different values remain. We then compute an "optimal invalid" schedule for a transformed instance. It is invalid because it allows preemption, migration and parallelization. However, in the end this invalid schedule allows us to compute a "good" feasible schedule for our instance.

The resource rounding is done with a linear grouping technique similar in spirit as the technique employed in [FdlVL81] for BIN-PACKING. We first sort the jobs from $\mathscr{J}'_{\text{tiny}}$ non-decreasingly by resource requirement. Let $\mathscr{J}'_{\text{tiny}} = \{j_1, \ldots, j_n\}$ be in this order. For $K := \lceil 1/\epsilon^2 \rceil$, we divide the jobs into K groups as follows: Figuratively, we take a schedule where the jobs

from $\mathcal{J}'_{\text{tiny}}$ are scheduled sequentially in the order from above, slice it into K intervals of equal length and define that $\mathcal{J}_i$ is the group of jobs that are completely contained in interval i for each $i = 1, \dots, K$. Group $\mathcal{J}_0$ is the set of jobs that are cut when defining the intervals. Formally, define

$$\mathcal{J}_i := \left\{ j_k \,:\, (i-1) \cdot p(\mathcal{J}'_{\text{tiny}})/K \leq \sum_{\ell=1}^{k-1} p(j_\ell) \leq i \cdot p(\mathcal{J}'_{\text{tiny}})/K - p(j_k) \right\}$$

for $i = 1, \dots, K$ and $\mathcal{J}_0 := \mathcal{J}'_{\text{tiny}} \setminus \bigcup_{i=1}^{K} \mathcal{J}_i$. By construction, we obtain the following properties of the set system. We skip the straightforward proof.

Lemma 4.2.15. *We have $|\mathcal{J}_0| \leq K$, $p(\mathcal{J}_i) \leq p(\mathcal{J}'_{\text{tiny}})/K$ for all $i = 1, \dots, K$, and for any two jobs $j \in \mathcal{J}_i$ and $j' \in \mathcal{J}_{i'}$ with $1 \leq i < i'$ it holds that $r(j) \leq r(j')$.*

Based on this grouping, we define a set of jobs $\tilde{\mathcal{J}}'_{\text{tiny}} := \{g_1, ..., g_{K-1}\}$ by setting $p(g_i) := p(\mathcal{J}'_{\text{tiny}})/K$ and $r(g_i) := \max\{r(j) \mid j \in \mathcal{J}_i\}$ for each $i \in \{1, ..., K-1\}$. For the resulting instance $\tilde{\mathcal{J}}' := \mathcal{J}'_{\text{large}} \cup \tilde{\mathcal{J}}'_{\text{tiny}}$, we later compute an "invalid" schedule. The jobs g_i are going to act as placeholders to fill in the jobs from $\mathcal{J}_i$ in the final solution later. The groups $\mathcal{J}_0$ and $\mathcal{J}_K$ are treated differently. They can be scheduled separately:

Lemma 4.2.16. *The jobs in $\mathcal{J}_0 \cup \mathcal{J}_K$ can be scheduled with makespan $O(1) \cdot \epsilon$.*

Proof. Because the jobs in $\mathcal{J}_0$ are tiny we have $p(\mathcal{J}_0) \leq K \cdot \frac{\gamma}{M} = O(1) \cdot \epsilon$. For $\mathcal{J}_K$, recall the resource consumption $\bar{R}$ from Section 4.2.1. Using the lower bound from Lemma 4.2.2 and the fact that the jobs are fat, we get

$$1 \geq OPT(\mathcal{J}') \geq \bar{R}(\mathcal{J}') \geq \bar{R}(\mathcal{J}'_{\text{tiny}}) \geq \beta \cdot p(\mathcal{J}'_{\text{tiny}}) \geq \beta \cdot K \cdot p(\mathcal{J}_K) \geq p(\mathcal{J}_K)/\epsilon.$$

Hence the total processing time of jobs from $\mathcal{J}_0 \cup \mathcal{J}_K$ is negligible. We conclude that if we schedule the jobs $\mathcal{J}_0 \cup \mathcal{J}_K$ sequentially on one machine, the makespan is $O(1) \cdot \epsilon$. □

We remark that this is the only point in the proof where we needed that the jobs we consider are fat. We now discuss how to compute the "invalid" helper schedule. We call it a *relaxed* schedule. In relaxed schedules, we allow jobs to be preempted, migrated, and executed in parallel. However the same rules for feasibility apply as for "real" schedules, i.e., each job gets assigned sufficient exclusive processing time on the machines, and the resource constraint is not violated. For simplicity of presentation, we refrain from a formal definition of relaxed schedules. We first prove the existence of a relaxed schedule for $\tilde{\mathcal{J}}'$.

Lemma 4.2.17. *If φ_2 is extendable for $\mathcal{J}'$, then it is extendable (as a relaxed schedule) for $\tilde{\mathcal{J}}'$.*

Proof. Let $\bar{\varphi}_2$ be an extension of φ_2 for $\mathcal{J}'$. Recall the illustration of slicing a sequential schedule for $\mathcal{J}'_{\text{tiny}}$ into K equally sized intervals. To construct a relaxed schedule for $\tilde{\mathcal{J}}'$, for each $i = 1, \dots, K-1$ we use the jobs $\mathcal{J}_{i+1}$ from interval $i+1$ as a template to fill in the job g_i. Note that this might include some fractions of jobs from that interval which are not included

in $\mathscr{J}_i$ but in $\mathscr{J}_0$. The processing times of these jobs are then split accordingly. This way, for each job g_i we have the "right" amount of processing time $p(g_i) = p(\mathscr{J}'_{\text{tiny}})/K$ available. Also the resource constraint stays satisfied because the resource requirement of job g_i is no more than the resource requirement of any job from interval $i+1$ being replaced. The jobs from the first group $\mathscr{J}_1$ are not used and are simply removed from the schedule. □

After proving the existence, we now show how to compute such a relaxed schedule for $\tilde{\mathscr{J}}'$. We again resort to linear programming. Note that in order to extend φ_2 we can use only resources "left over" by the large jobs $\mathscr{J}'_{\text{large}}$. Based on φ_2, for each frame $F \in \mathscr{F}$ let $r(F)$ denote the amount of resources available for non-large jobs, and let $m(F)$ denote the number of machines available. Note that these quantities are the same throughout a frame as the schedule φ_2 is δ-atomic. We will use these remaining resources to schedule tiny jobs of the instance.

For each schedule extending φ_2, every possibility of jobs $g_1, \ldots, g_{K-1}$ being simultaneously active during frame $F \in \mathscr{F}$ can be described by a characteristic vector $\chi \in [m]^{K-1}$ such that $\sum_{i=1}^{K-1} \chi_i r(g_i) \leq r(F)$ and $\sum_i \chi_i \leq m(F)$. For each job g_i, the entry χ_i specifies the number of machines used for executing job g_i in parallel. Let $\mathscr{C}(F)$ be the set of all such vectors. We call a vector $\chi \in \mathscr{C}(F)$ a *job configuration*. This allows us to formulate a "configuration-LP" that packs job configurations to frames and ensures that all tiny jobs are covered. Denote by CONF-LP the following linear program.

$$\sum_{F \in \mathscr{F}} \sum_{\chi \in \mathscr{C}(F)} \chi_i x_{F\chi} \geq p(g_i) \qquad \forall i = 1, \ldots, K-1 \tag{4.4}$$

$$\sum_{\chi \in \mathscr{C}(F)} x_{F\chi} \leq \delta \qquad \forall F \in \mathscr{F} \tag{4.5}$$

$$x_{F\chi} \geq 0 \qquad \forall F \in \mathscr{F}\ \forall \chi \in \mathscr{C}(F)$$

The variable $x_{F\chi}$ models for how much time configuration χ should be used within frame F. The constraints of type (4.4) ensure that each job g_i gets assigned enough processing time. The constraints of type (4.5) ensure that the total amount of configurations assigned to each frame does not exceed the frame size, i.e., they ensure that we have a valid packing.

Observe that a feasible relaxed schedule extending φ_2 gives rise to a solution of CONF-LP in a straightforward way. Hence with Lemma 4.2.17 we know that CONF-LP has a feasible solution. As mentioned before, a basic fact from linear optimization is that if a linear program has a feasible solution, then there is an extreme point solution, i.e. a solution whose number of nonzero variables is bounded by the number of constraints (ignoring non-negativity constraints). An extreme point solution hence fulfills the properties of the following lemma:

Lemma 4.2.18. *In polynomial time we can compute a solution to CONF-LP with at most $K + |\mathscr{F}|$ non-zero variables.*

It remains to construct a non-relaxed schedule φ'_2 for $\mathscr{J}'$ from the solution to CONF-LP. We

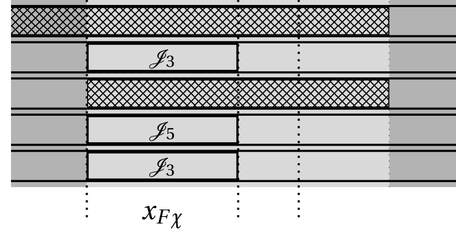

Figure 4.4: Illustration of the construction of a non-relaxed schedule from an LP-solution.

now describe this construction. It is also illustrated in Figure 4.4. We partition each frame $F \in \mathscr{F}$ into sub-frames, each sub-frame corresponds to a positive variable $x_{F\chi}$ and has length $x_{F\chi}$. The packing constraints (4.5) ensure that we can do that. Now for each sub-frame and each $i \in [K]$, create χ_i many machine slots (assign them to free machines greedily) and reserve them for jobs of group $\mathscr{J}_i$. The machine slots created in this way act as placeholders and, to avoid confusion, we will refer to them as *placeholder slots*. Our construction ensures that if we pack jobs of group $\mathscr{J}_i$ arbitrarily to its reserved placeholder slots, we will not violate the resource requirement (as by Lemma 4.2.15, the resource requirement of all jobs from $\mathscr{J}_i$ is at most $r(g_i)$). As the covering constraints (4.4) are satisfied, the total amount of execution time reserved for each group $\mathscr{J}_i$ is at least $p(g_i)$ which by definition and Lemma 4.2.15 is at least $\sum_{j\in\mathscr{J}_i} p(j)$.

Now for each group $\mathscr{J}_i$, $i = 1 \dots K-1$ and each job $j \in \mathscr{J}_i$, select an arbitrary placeholder slot reserved for group $\mathscr{J}_i$ which has a positive amount of space left and assign j to it. We stress that we assign j even if the positive amount of space is less than the jobs processing time. By the observations from above, it is clear that this algorithm manages to assign all jobs. However, it might produce an infeasible solution as some placeholder slots might be over-packed. We can repair this as follows: For each sub-frame (i.e., for each positive variable in the LP-solution), and for each placeholder slot belonging to this sub-frame, pick the job added last and remove it. Now the placeholder slots are not over-packed anymore, i.e., the resulting schedule is feasible. For the removed jobs, observe that those that are taken from the same sub-frame can be scheduled in parallel. Hence, because they are tiny, we can schedule them separately in a time-frame of γ/M time-units. As there are at most $|\mathscr{F}| + K$ many nonzeros the LP solution due to Lemma 4.2.18, we conclude that the increase of the makespan is at most

$$\gamma/M \cdot (|\mathscr{F}| + K) \leq \gamma/\epsilon^{-3} \cdot \left(\frac{1+\epsilon}{\epsilon \cdot \gamma/2} + 1/\epsilon^2 + 2\right) = O(1) \cdot \epsilon.$$

Hence, in summary we get the following result:

Lemma 4.2.19. *Given φ_2, in polynomial time we can compute a schedule φ_2' for $\mathscr{J}'$ with $T(\varphi_2') \leq 1 + O(1) \cdot \epsilon$. Moreover we have $A(\varphi_2') \leq O(1) \cdot \frac{1}{\epsilon^2 \gamma}$.*

Proof. The makespan increase due to the above procedure, as well as the length of the schedules for $\mathscr{J}_0$ and $\mathscr{J}_K$, is bounded by $O(1) \cdot \epsilon$. To see the bound on the activation points, assume

that the jobs in each slot of each sub frame are placed in non-increasing order of their resource requirement. Then only one new activation point can only be introduced per sub-frame from the construction above. As φ_2 has at most $O(1)\cdot\frac{1}{\epsilon\cdot\gamma}$ activation points by Lemma 4.2.14 and the number of sub-frames is bounded by $|\mathscr{F}|+K = O\left(\frac{1+\epsilon}{\epsilon\cdot\gamma/2}+1/\epsilon^2\right)$, the bound on the activation points follows. □

Step 2b: Adding tiny thin jobs.

Let φ_1 and φ_2' be the schedules obtained due to Lemma 4.2.12 and Lemma 4.2.19, respectively. Observe that if we concatenate φ_1 and φ_2', we obtain a schedule of makespan $2+O(1)\cdot\epsilon$ that schedules all jobs from $\mathscr{J}$ except for the tiny thin ones. The number of activation points of this schedule is $O(1)\cdot 1/(\epsilon^2\gamma)$.

We use the list scheduler from Section 4.2.1 to complete the schedule. Applying Lemma 4.2.7 in this situation, we can set $\tau := \gamma/M$. Hence we obtain a full schedule of makespan

$$\begin{aligned} T(\varphi) &\le \max\left\{2+O(1)\cdot\epsilon, \bar{P}(\mathscr{J})+\frac{1}{1-\beta}\bar{R}(\mathscr{J})+O(1)\cdot\frac{1}{\epsilon^2\gamma}\gamma/M\right\} \\ &\le \max\left\{2+O(1)\cdot\epsilon, 1+\frac{1}{1-\epsilon}+O(1)\cdot\frac{1}{\epsilon^2\gamma}\gamma/\epsilon^{-3}\right\} \\ &= 2+O(1)\cdot\epsilon. \end{aligned}$$

Putting all the steps together, we obtain the $(2+\epsilon)$-approximation algorithm.

Theorem 4.2.20. *There is a $(2+\epsilon)$-approximation algorithm for the non-preemptive resource constrained scheduling problem on identical machines.*

Proof. We recap all the steps of the algorithm. Given an instance $\mathscr{J}$ with $OPT\le 1$, with the constants defined throughout the section we partition the instance $\mathscr{J}=\mathscr{J}_1\cup\mathscr{J}_2$ as indicated in Lemma 4.2.9. $\mathscr{J}_2$ can be scheduled separately with the list scheduler and has a makespan of $O(1)\cdot\epsilon$. $\mathscr{J}_1$ is (γ, M)-restricted. We now compute a δ-atomic schedule φ_1 of makespan $1+\epsilon$ for all but a constant amount of large jobs by enumerating all templates from Lemma 4.2.11 and applying the algorithm from Lemma 4.2.12. It is successful for at least one template. For the remaining large jobs not scheduled in φ_1, we compute a set of candidate schedules as explained in Lemma 4.2.14. For each of them, we try to extend it by adding all the tiny fat jobs. As at least one candidate is extendable, by Lemma 4.2.19, this is successful for at least one candidate and we obtain a schedule φ_2' of makespan $1+O(1)\cdot\epsilon$. Concatenating φ_1 and φ_2', we obtain a schedule of makespan $2+O(1)\cdot\epsilon$ that schedules all large jobs and all tiny fat jobs. We add the remaining tiny thin jobs greedily using the list scheduler. As seen above, the resulting schedule has makespan $2+c\cdot\epsilon$ for some constant $c=O(1)$. To compute a schedule of makespan $2+\epsilon$, set $\epsilon' := \epsilon/c$ and run the algorithm with parameter ϵ' instead of ϵ.

Finally to turn this algorithm into a $(2+\epsilon)$-approximation algorithm, we use a standard binary

search framework. We sketch it here, but refer to a presentation in full detail to Section 4.2.4. First we find suitable lower and upper bounds L and U on the optimal makespan. Then set $M := \frac{U+L}{2}$. This is our "guess" on the optimal makespan. Rescale all processing times by $1/M$, then apply our algorithm. If it returns a feasible schedule of makespan at most $2+\epsilon$ for the scaled instance, we know that $OPT \leq (2+\epsilon)\cdot M$ and can set $U := M$. Otherwise, if our algorithm fails, we know that $OPT \geq M$ and can set $L := M$. We iterate this procedure until $U - L$ is below some suitable margin. The total number of iterations is polynomial in the input size because the length of the interval $[L, U]$ decreases exponentially. □

4.2.3 Polynomial Time Approximation Schemes for Special Cases

We can improve the approximation guarantee of the previous algorithm if we restrict ourselves to special cases. We will deal with two cases. The first is that m, the number of machines, is upper bounded by a constant. The second case is that the number of *different* resource requirements in the instance $\mathcal{J}$ is upper bounded by a constant. In both cases we will modify the algorithm from above to get a polynomial time approximation scheme, i.e., for each constant $\epsilon > 0$ we can get a $(1+\epsilon)$-approximation algorithm.

Essentially there are two steps of the algorithm that need to be improved. The first one is the way we deal with large jobs: Instead of creating two schedules of makespan $1+\epsilon$ and concatenating them, we need to treat all large jobs together. The second issue is the use of the list scheduler to schedule the thin tiny jobs: No matter how we tweak the parameters, this algorithms performance guarantee will not get better than $2+\epsilon$. Both issues will be addressed for both special cases, but in different ways. As before, we assume the instances $\mathcal{J}$ we consider have an optimal solution of makespan at most 1, i.e., $OPT(\mathcal{J}) \leq 1$.

Constant number of machines

We now assume that all instances for the resource constrained scheduling problem have the property that there is a universal constant $C > 0$ so that the number of machines m is upper bounded by C. We also assume (without loss of generality) that $\frac{1}{\epsilon} \geq C$. Adapting the way on how we treat the large jobs is very easy using the following insight. If there is a schedule of makespan at most 1, how many large jobs can it have? As every large job has processing time at least γ, an optimal schedule can process at most $1/\gamma$ many jobs per machine. As there are at most C machines, there are at most C/γ, i.e., *constantly many*, large jobs. Hence to treat the large jobs, we simply skip step 1a. Lemma 4.2.14 guarantees that in polynomial time we can compute a candidate set of feasible δ-atomic schedules of makespan $1+\epsilon$ for all large jobs $\mathcal{J}_{\text{large}}$ of the instance, where at least one of them is extendable to a full schedule for $\mathcal{J}$.

To deal with the tiny jobs, we need to recall why the distinguishment of *fat* and *thin* jobs was made in the first place. Recall that in the process of dealing with fat jobs, we removed two subsets of jobs and scheduled them separately (namely the sets $\mathcal{J}_0$ and $\mathcal{J}_K$). We argued

that they can be scheduled separately using only a negligible, i.e., $O(1)\cdot\epsilon$ amount of time. Particularly, the argument for $\mathcal{J}_K$ was that as the jobs are fat, only $1/\beta$, i.e., constantly many, of them can run in parallel. Now, only constantly many of them can run in parallel anyway because we have only at most C many machines. Hence we do not need to distinguish between fat and thin jobs anymore and treat all tiny jobs as described in Step 2a. We need however a new proof for Lemma 4.2.16.

Lemma 4.2.21. *Consider the linear grouping process as described in Section 4.2.2, where the linear grouping is done with all tiny jobs. The jobs in $\mathcal{J}_0 \cup \mathcal{J}_K$ can be scheduled with makespan $O(1)\cdot\epsilon$.*

Proof. The proof for the jobs from $\mathcal{J}_0$ is as in Lemma 4.2.16: There are at most K many jobs, so their total processing time is a most $p(\mathcal{J}_0) \le K\cdot\frac{\gamma}{M} \le O(1)\cdot\epsilon$. For $\mathcal{J}_K$, recall the load bound $\bar{P}$ from Section 4.2.1. Using that at most C jobs run in parallel, we get that

$$1 \ge OPT(\mathcal{J}) \ge \bar{P}(\mathcal{J}) \ge \bar{P}(\mathcal{J}_{\text{tiny}}) \ge p(\mathcal{J}_{\text{tiny}})/C \ge K/C\cdot p(\mathcal{J}_K) \ge p(\mathcal{J}_K)/\varepsilon.$$

We conclude that if we schedule the jobs $\mathcal{J}_0 \cup \mathcal{J}_K$ sequentially on one machine, the makespan is $O(1)\cdot\epsilon$. □

Thus for the special case of constant number of machines, we have a PTAS, simply by using only a subset of components from the main algorithm.

Theorem 4.2.22. *For any constant C and $\epsilon > 0$, there is a $(1+\epsilon)$-approximation algorithm for all instances $\mathcal{J}$ with $m \le C$.*

Proof. We follow the steps of the $(2+\epsilon)$-approximation algorithm presented in Section 4.2.2. In particular, using the same constants as before, we again restrict to the case of (γ, M)-restricted instances with $OPT(\mathcal{J}) \le 1$. As observed above, the total number of large jobs in $\mathcal{J}$ is constant. Hence with Lemma 4.2.14 we can compute a candidate set of feasible schedules for $\mathcal{J}_{\text{large}}$. One of them is extendable to a full schedule for $\mathcal{J}$ of makespan $1+\epsilon$.

We then apply the algorithm from Step 2a for all tiny jobs. Lemma 4.2.21 and Lemma 4.2.19 guarantee that we will find a schedule for $\mathcal{J}$ of makespan $1+O(1)\cdot\epsilon$. □

Constant number of different resource requirements

We now assume that there is a universal constant $C > 0$ so that for each instance $\mathcal{J}$, there are at most C different resource requirements: There are numbers $R_1,\ldots,R_C$ so that $r(j) = R_\ell$ for some $\ell = 1,\ldots,C$ for all $j \in \mathcal{J}$. To deal with large jobs, we proceed as follows. Recall the *templates* from Step 1a. We noted that given a feasible template, it is NP-hard to assign the jobs to the machine slots in the template so that we get a feasible schedule. In this special case we can circumvent this as follows. We define a *colorful* machine slot as a pair $(s,\ell) \in \mathcal{S} \times [C]$.

I.e., a colorful machine slot is a "regular" machine slot s that is reserved for a job with resource requirement R_ℓ. Analogous, a *colorful template* is a set of $|\mathcal{J}|$ colorful machine slots without a job assignment. It is feasible if there is a feasible schedule that uses the colorful machine slots from the templates for jobs of corresponding resource requirement. Unlike the regular templates, it is now easy to determine whether a template is feasible and to compute a feasible schedule from a feasible (colorful) template: If a template is feasible, the following greedy algorithm successfully computes a feasible schedule: While there is an unassigned job, choose the one of largest processing time and assign it to any machine slot of its resource requirement whose length is sufficient. We skip the straightforward proof. Similar to Lemma 4.2.11, in polynomial time we can enumerate a set of candidate templates:

Lemma 4.2.23. *In polynomial time, we can compute a set of colorful templates. At least one of them is feasible and extendable.*

Proof. This proof is essentially by the lines of the proof of Lemma 4.2.11. For a colorful machine slot, we can choose the resource group, the start and the end interval. Also there are at most $|\mathcal{F}|$ many large jobs on each machine. Hence there are at most $Q := \left(C \cdot \binom{|\mathcal{F}|}{2}\right)^{|\mathcal{F}|}$ many combinations of colorful machine slots for a fixed machine that could lead to a feasible schedule. We can now describe a template by specifying for each possibility, how many of the m machines use it. We conclude that (up to permutation of machines), there are at most m^Q many colorful templates that might be feasible. We can enumerate them all. One of them corresponds to the feasible δ-atomic schedule from Lemma 4.2.10 and is hence feasible and extendable. □

This shows that in polynomial time, we can compute a schedule φ_1 of makespan $1+\epsilon$ for *all* large jobs $\mathcal{J}_{\text{large}}$. Moreover, φ_1 is extendable, i.e., we can add all tiny jobs without increasing the makespan. It remains to deal with the tiny jobs. As in the case of constant number of machines, we will not distinguish between thin and fat jobs. Instead we will modify the grouping procedure. Recall that in Step 2a, we grouped the tiny fat jobs into $K+1$ groups $\mathcal{J}_0, \ldots, \mathcal{J}_K$ and rounded the resource requirement. Now we instead define C many groups $\mathcal{J}_1, \ldots, \mathcal{J}_C$ by resource requirement, i.e.,

$$\mathcal{J}_\ell := \left\{ j \in \mathcal{J}_{\text{tiny}} \, : \, r(j) = R_\ell \right\}$$

Analogous to Step 2a, we now construct a helper instance $\widetilde{\mathcal{J}_{\text{tiny}}} := \{g_1, \ldots, g_C\}$. We set $p(g_\ell) := p(\mathcal{J}_\ell)$ and $r(g_\ell) := R_C$. We observe that as φ_1 is extendable, φ_1 is also extendable as a relaxed schedule for $\tilde{\mathcal{J}} := \mathcal{J}_{\text{large}} \cup \widetilde{\mathcal{J}_{\text{tiny}}}$. The proof is a simplified version of the proof for Lemma 4.2.17 and we omit the details. Now we can argue by the lines of the description of Step 2a that CONF-LP finds a good extreme point solution which we can turn into a full schedule for $\mathcal{J}$ of makespan $1 + O(1) \cdot \epsilon$. We conclude:

Theorem 4.2.24. *For any constant C and $\epsilon > 0$, there is a $(1+\epsilon)$-approximation algorithm for all instances $\mathcal{J}$ where the number of different resource requirements is bounded by C.*

4.2.4 A Binary Search Framework

Recall that for the $(2+\epsilon)$-approximation scheme shown in Section 4.2.2, we actually only presented an algorithm that for any instance $\mathscr{J}$ with $OPT(\mathscr{J}) \leq 1$, finds a schedule of makespan at most $2+\epsilon$. We then claimed that this algorithm can be turned into a $(2+\epsilon)$-approximation scheme using a standard binary search framework. For the sake of being self-contained, we now describe this framework in more detail.

Given an instance $\mathscr{J}$, first we compute suitable lower and upper bounds L and U on the optimal makespan. For example, we can set $L := \bar{P}(\mathscr{J}) = \frac{1}{m}\sum_{j\in\mathscr{J}} p(j)$ and $U := \sum_{j\in\mathscr{J}} p(j)$. One could find even better bounds, but they serve the purpose. Now while the difference between U and L is above some margin, we repeat the following procedure. The goal is to decrease the distance of U and L while keeping the invariant that $(2+\epsilon)\cdot U$ and L are upper/lower bounds for the optimal makespan (for some constant c). Set $M := \frac{U+L}{2}$, scale all processing times of the jobs from $\mathscr{J}$ by $\frac{1}{T}$, and run the algorithm from Section 4.2.2. If the algorithm returns a schedule of makespan at most $2+\epsilon$, we can easily translate it into a schedule of makespan $(2+\epsilon)T$ for the unscaled instance. Thus we can set $U := M$ and the invariant still holds. Otherwise, if the algorithm fails to compute such a schedule, we know that the optimum makespan for the scaled instance is at least 1. This translates to the fact that in the original instance, the optimal makespan is at least T. Hence we can safely set $L := M$ without invalidating the invariant.

When do we stop? Let $g := \gcd_{j\in\mathscr{J}} p(j)$. We claim that the optimal makespan is an integer multiple of g. Assume otherwise, i.e., consider an optimal schedule whose makespan is not a multiple of g. Then there must be a job that does not start at at multiple of g. Amongst all such jobs, let j^* be one with the smallest starting time. We can then round down the starting time of j^* to the nearest multiple of g without changing feasibility of the schedule, because all starting and ending times before are multiples of g. Iterating the argument leaves us with a schedule where all starting and ending times are multiples of g. Moreover this construction did not increase the makespan. Hence the schedule is still optimal. Also as all ending times are multiples of g, the makespan is a multiple of g. A contradiction.

As the optimal makespan is an integer multiple of g, we know that we can stop if $U - L < g$. To see this, let $\tilde{L}$ be L rounded up to the nearest multiple of g. We know that $\tilde{L} \leq OPT \leq (2+\epsilon)\cdot U$. If $\tilde{L} = OPT$, then our algorithm on input $\mathscr{J}$ scaled by $\tilde{L}$ will compute a schedule of makespan at most $(2+\epsilon)\cdot OPT$. Otherwise, by the termination criterion we have that $OPT \geq \tilde{L} + g > U$. As U always corresponds to a successful run of the algorithm, we have a schedule of makespan at most $(2+\epsilon)\cdot U \leq (2+\epsilon)\cdot OPT$. In both cases we have a $(2+\epsilon)$-approximation. See Listing 3 for a more formal description of the algorithm.

Lemma 4.2.25. *Given a makespan minimization instance $\mathscr{J}$ and oracle access to the $(2+\epsilon)$-decision algorithm ALG from Section 4.2.2, the algorithm from Listing 3 computes a schedule of*

Algorithm 3 The binary search framework.

Input: $\mathscr{J}$**,** L**,** U**,ALG**
$\gamma \leftarrow \gcd\{p(j) : j \in \mathscr{J}\}$
$\varphi \leftarrow ALG(\mathscr{J}, U)$.
while $U - L > \gamma$ **do**
  $M \leftarrow \frac{U+L}{2}$
  $\varphi' \leftarrow \mathrm{ALG}(\mathscr{J}, M)$
  if $\varphi' =$ **FAIL then**
    $L \leftarrow M$
  else
    $U \leftarrow M$
    $\varphi \leftarrow \varphi'$
  end if
end while
$\tilde{L} \leftarrow L$ **rounded up to nearest multiple of** γ**.**
return best of $\{\varphi, \mathrm{ALG}(\mathscr{J}, \tilde{L})\}$

makespan $(2+\epsilon)\cdot OPT(\mathscr{J})$ *Its running time is*

$$O\big(\big(\log(U-L) + \log(1/\gamma\big)\cdot T_{ALG}(\mathscr{J})\big),$$

where $T_{ALG}(\mathscr{J})$ *is the running time of the algorithm from Section 4.2.2 and* $\gamma = \gcd\{p(j) : j \in \mathscr{J}\}$.

Proof. With the discussion from above, it is obvious that the algorithm computes a schedule of makespan $(2+\epsilon)\cdot OPT(\mathscr{J})$. It remains to show the bound on the running time. Let L_k and U_k be the bounds U and L after the k-th execution of the while loop. In particular, let L_0 and U_0 be the original bounds. Observe that $U_k - L_k = \frac{1}{2}(U_{k-1} - L_{k-1})$ for all $k \in \mathbb{N}$, i.e., the interval $[L, U]$ is halfed in each iteration of the loop. The algorithm terminates if $U_k - L_k \leq \gamma$, i.e., if

$$\gamma \geq 2^{-k}(U-L)$$

or equivalently

$$k \geq \log(1/\gamma) + \log(U-L)$$

□

It remains to argue that the running time is polynomial in the encoding length of the input. Each processing time $p(j)$ is a rational number, therefore its encoding is given by two numbers $a_j, b_j \in \mathbb{Z}$ so that $p(j) = a_j/b_j$. Let $B := \prod_j b_j$. Observe that $U = A/B$, where A is a number whose absolute value is bounded by $n\cdot m\cdot B\cdot \max_j |a_j|$. We conclude that the binary encoding length of both A and B, and therefore U, is bounded by a polynomial in the encoding length of the original instance. The same trivially holds for L. Also for γ, an upper bound for the absolute of its denominator is given by B and an upper bound for the absolute value of its

nominator is given by $n \cdot m \cdot B \cdot \max_j |a_j|$. We conclude that the Algorithm from Lemma 4.2.25 has polynomial running time.

4.2.5 Inapproximability

At the end of this section we ask for lower bounds on the approximability of the resource constrained scheduling problem. We show that the problem is NP-hard to approximate with a factor better than $\frac{3}{2}$. This shows that the computational complexity of the problem actually increases when adding the resource constraint, as there is a PTAS for the basic problem [HS87]. The result is established via a straightforward reduction from the NP-hard PARTITION problem. Here one has given a set of n items with item sizes $a_1, \dots, a_n \in \mathbb{N}$. The question is whether these items can be partitioned into two sets, so that the total item size of each set is the same. For the reduction, let $B := \frac{1}{2} \sum_{i=1}^{n} a_i$. For each item a_i, we introduce a job of processing time 1 and resource requirement a_i / B. Observe that there is a schedule of makespan 2 if and only if the items can be partitioned into two sets of equal size B. Also observe that if this partitioning does not exist, the shortest possible non-preemptive schedule has makespan 3. We conclude:

Theorem 4.2.26. *For the resource constrained scheduling problem there is no approximation algorithm with an approximation factor better than* $3/2$ *unless* $P = NP$.

4.3 Resource Constrained Scheduling on Unrelated Machines

We now focus on the case of unrelated machines. To this end, we alter notation slightly to account for this more general model. Given an instance $\mathcal{J}$ with n jobs and m machines, execution times $p(j,k)$ and resource requirements $r(j,k)$ are now given for each job/machine pair (j,k). It means that if a job $j \in \mathcal{J}$ is assigned to machine k, it requires a processing time of $p(j,k)$ and has a resource requirement of $r(j,k)$ when active. The definition of schedules as maps that assign machine slots of suitable length to the jobs remains unchanged. As before, the goal is to compute a feasible schedule of minimum makespan. We present a 3.75-approximation algorithm. It computes a non-preemptive schedule. However in the analysis we compare its performance only to optimal preemptive schedules, which implies that this algorithm is a 3.75-approximation algorithm both in the preemptive and non-preemptive setting.

The algorithm works in two phases. In the first phase, only an assignment of jobs to machines is computed. Then in in phase two we compute the actual schedule based on the machine assignment. The first phase essentially consists of using an algorithm by Shmoys and Tardos [ST93] that computes a schedule for a variation of makespan minimization scheduling they call *scheduling with costs*, invoked with the “right” parameters. The schedule generation in the second phase is done with a list scheduler.

The presentation of the algorithm is structured as follows. We first introduce the *scheduling problem with costs* and state the approximation algorithm by Shmoys and Tardos for this

problem (Section 4.3.1). We then present a 4-approximation algorithm in Section 4.3.2. Here we explain the two phases in more detail and analyze the approximation guarantee of the algorithm. Finally in Section 4.3.3 we discuss a variation of the algorithm and show that it is a 3.75-approximation algorithm.

As mentioned previously, there is a 3.75-approximation algorithm by Grigoriev, Sviridenko and Uetz [GSU07] from 2007 for the even more general problem of scheduling jobs with resource dependent processing times. The 4-approximation presented in Section 4.3.2 was found independently. Nevertheless, the basic idea of having a two-phase approach is quite similar: also Grigoriev et al. first compute a machine assignment and then the actual schedule. As they deal with a more general problem, the subproblem to solve in the first phase is also more general, and they provide their own algorithm rather than being able to use [ST93] as a black-box. The subproblem that remains in the second phase is similar to ours and they also employ a list scheduler. Details however differ from our list scheduler. While our 4-approximation from Section 4.3.2 is independent work, the adaptation to turn it into a 3.75-approximation presented in Section 4.3.3 is based on insights from [GSU07].

4.3.1 Scheduling on Unrelated Parallel Machines with Costs

For the moment, forget about the resource constraint and consider the basic problem of scheduling jobs on unrelated parallel machines, minimizing the makespan. As mentioned previously, Lenstra, Shmoys and Tardos [LST90] constructed a 2-approximation algorithm for this problem. More precisely, given an instance and makespan bound T, their algorithm either asserts that there is no schedule of makespan at most T, or provides a schedule of makespan $2 \cdot T$. The main idea is to consider some linear programming relaxation for this problem. Based on an optimal extreme point solution, a rounding technique is employed to find a feasible integer solution of required quality. The rounding technique is based on finding a matching in a certain bipartite graph. Given the algorithm to decide this decision variant (with input T), a standard binary search framework as already explained in Section 4.2.4 can be used to turn it into a 2-approximation algorithm.

Shmoys and Tardos also studied a generalization of the problem, which they call the *scheduling problem with costs*. Here, in addition to the processing times, one gets costs $c(j,k)$ as input for each job/machine pair (j,k). When a job j is assigned to machine k, it incurs a cost of $c(j,k)$. Given a makespan bound T, the goal is to find a schedule of makespan at most T of minimum cost. Shmoys and Tardos designed a “2-approximation” algorithm for this problem [ST93]. The exact statement is as follows:

Theorem 4.3.1 ([ST93])**.** *Consider an instance of the scheduling problem with costs for which there is a machine assignment with makespan at most T and total cost C. Let*

$$\tau := \max_{j,k\,:\,p(j,k)\leq T} \{p(j,k)\}$$

denote the largest processing time of any job on any machine. Given the instance and the value of T as input, the polynomial time algorithm in [ST93] computes a machine assignment of makespan at most $T+\tau$ and cost at most C.

The technique employed to prove the result is essentially a generalization of the LP-rounding technique via finding a matching in a certain helper graph as described above.

4.3.2 A 4-Approximation Algorithm

We now come back to the resource constrained scheduling problem on unrelated machines, and describe our 4-approximation algorithm. As mentioned before, the algorithm works in two phases. In the first phase, an assignment of jobs to machines is computed. Afterwards, based on this assignment, in phase two we compute the actual schedule using a list scheduler.

Let $\sigma : \mathscr{J} \to [m]$ denote a machine assignment. We can lower bound the makespan of every feasible schedule using *this* assignment as follows. Note that the assignment σ completely determines the processing time and resource requirement of a job. For notational convenience we set $p(j) := p(j,\sigma(j))$ and $r(j) := r(j,\sigma(j))$. We can now derive a lower bound on the makespan in terms of total processing time and *resource consumption* similar to the case of identical machines in Lemma 4.2.2. We define the *resource consumption* of a job j (under σ) as the resource/processing product $r(j)\cdot p(j)$.

Lemma 4.3.2. *Let φ be a schedule using machine assignment σ. Then*

$$T(\varphi) \geq \bar{R}(\mathscr{J}) := \sum_{j\in\mathscr{J}} r(j)\cdot p(j).$$

Proof. As the resource usage of φ is limited to 1 at all times, the schedule can "process" only one unit of resource consumption per time unit. We conclude that the schedule needs at least $\bar{R}(\mathscr{J})$ time units to "process" the total resource consumption of $\mathscr{J}$, which implies the makespan bound. □

Another simple and obvious lower bound on the makespan of a schedule using σ is the maximum load of a machine, i.e., the sum of execution times of jobs assigned to it. We skip the proof for triviality.

Lemma 4.3.3. *Let φ be a schedule using machine assignment σ. Then*

$$T(\varphi) \geq \max_{k\in[m]} \sum_{j\,:\,\sigma(j)=k} p(j).$$

These bounds give an indication on how to design the first phase of our algorithm: The machine assignment should be in such a way that both the machine load and the total resource consumption is "low". We take this into account when computing the machine assignment in

the first phase. These bounds are also crucial in the analysis of the approximation guarantee in the second phase.

Phase 1: Assigning jobs to machines

As just explained, our goal for phase one is to compute a machine assignment σ for which both resource consumption and processing load is low. To do so, we construct a suitable instance for the *scheduling problem with costs* introduced in Section 4.3.1. We will use the algorithm from Theorem 4.3.1 as a black-box to compute a machine assignment. To that end we model the costs as the resource consumption of the jobs: for each job/machine pair (j,k) we set

$$c(j,k) := r(j,k) \cdot p(j,k).$$

Reversing the argument from Lemma 4.3.2, we can upper bound the cost of any schedule of makespan T:

Lemma 4.3.4. *If there is a schedule of makespan T for the resource constrained scheduling problem, then there is a solution to the scheduling problem with costs of makespan T and cost at most T.*

Proof. As the schedule processes only one unit of resource consumption per time-unit, and it is finished at time T, the total resource consumption of the instance cannot be more than T. By construction, this implies that the cost of that schedule is at most T. □

With the properties of the algorithm of Shmoys and Tardos from Theorem 4.3.1 we can then conclude:

Corollary 4.3.5. *If there is a schedule of makespan T for the resource constrained scheduling problem, the Shmoys-Tardos algorithm determines a machine assignment of the jobs with machine load at most $(T+\tau)$ and total resource consumption at most T.*

Proof. By construction and by Lemma 4.3.4, if there is a schedule of makespan T for the resource constrained scheduling problem, there is a solution of makespan T and cost at most T for the scheduling problem with costs. Hence the algorithm from Theorem 4.3.1 finds a solution as stated. □

It remains to find the right T. Our goal is to find a machine assignment with the properties from Corollary 4.3.5 for some $T \le OPT(\mathscr{J})$. This is done using the binary search framework explained in Section 4.2.4: for all $T \ge OPT(\mathscr{J})$, Corollary 4.3.5 guarantees that the algorithm succeeds. Hence using suitable lower and upper bounds on T, e.g., $\min p(j,k)$ and $\sum p_{j,k}$ we can find the desired T via binary search in polynomial time.

Corollary 4.3.6. *Let*

$$\tau := \min\left\{OPT(\mathscr{J}), \max_{j,k} p(j,k)\right\}.$$

In polynomial time we can compute a machine assignment σ with machine load at most $(OPT(\mathscr{J}) + \tau)$ and total resource consumption at most $OPT(\mathscr{J})$.

Phase 2: Computing the schedule

Based on the machine assignment σ from Corollary 4.3.6 we now want to compute a schedule. To simplify notation, we define processing times and resource requirements for each job as implied by the machine assignment σ determined in Phase 1:

$$p(j) := p(j,\, \sigma(j)) \qquad r(j) := r(j,\, \sigma(j)) \;\; \forall j \in \mathscr{J}.$$

The schedule is generated using a greedy list scheduler. We sort the jobs non-increasing by their resource requirement $r(j)$ and create an ordered list

$$\mathscr{L} = (j_1, j_2, \ldots, j_n) \tag{4.6}$$

of all jobs with the property that for any two jobs j_a, j_b with $a < b$ we have $r(j_a) \geq r(j_b)$. The list defines job priorities for our phase-two algorithm, with the leftmost entry (i.e., the one with the highest resource requirement) having best priority and the rightmost entry having least priority. The list scheduler always assigns the job j with the best priority that is ***available*** at given time t, meaning that the following two conditions must be satisfied:

(a) Scheduling the job at time t does not lift the resource usage above 1.

(b) The job j has to be ***ready***. We call a job j ready at time t, if every job j' with $\sigma(j) = \sigma(j')$ and $r(j') > r(j)$ has finished at time t. (This can be seen as implicit precedence constraints between jobs on the same machine which enforce that jobs assigned to the same machine are always processed in the order given by their resource requirement.)

The full algorithm is presented in Listing 4.

The analysis

We first show that Algorithm 4 *runs in polynomial time*. As listed, Phase 1 assumes that the value of T is known and makes only one call to the procedure of Theorem 4.3.1. As discussed above, we can compute this number T via binary search using polynomially many calls to this polynomial-time procedure of Theorem 4.3.1. For the second phase, the **while** loop is executed at most once for each job completion, i.e., n times; during each iteration, the inner **for** loop can clearly be implemented to execute in polynomial time.

Algorithm 4 Scheduling resource-constrained jobs

```
{Phase 1: assign jobs, assuming minimum makespan T ≤ OPT(𝒥) is known}
σ ← ST({p(j,k)}_(n×m), {r(j,k)·p(j,k)}_(n×m), T)
{Phase 2: generate the schedule}
ℒ ← (j_1, j_2, ..., j_n) {as defined in (4.6)}
t ← 0
while ℒ has unscheduled jobs do
  for ℓ = 1 to n do
    if j_ℓ is unscheduled and available at time t then
      Schedule job j_ℓ on processor σ(j_ℓ) at time t by allocating suitable machine slot
    end if
  end for
  t ← next finishing time of a job.
end while
```

It is obvious that Algorithm 4 computes a feasible schedule. It remains to show the claimed approximation guarantee, i.e., that the schedule has makespan at most $4 \cdot OPT(\mathscr{J})$. Define

$$\beta := \min\left\{\frac{1}{2}, \max_{j,k} r(j,k)\right\}.$$

We will show that the schedule computed by our algorithm consists of two regions: In the first region, the resource usage is always high. More precisely, the resource usage is always at least $1-\beta$. In the second region however, the resource usage is always below $1-\beta$. As a consequence we can show that in the second region, there are no idle times due to resource limitations. In the end, this allows us to bound the total makespan of the computed schedule by separately computing the makespan of the two regions, using the resource consumption bound from Corollary 4.3.5 for the first region, and the processing load bound for the second region.

We start with a basic observation regarding the structure of this schedule:

Lemma 4.3.7. *Consider a schedule computed by the algorithm. If job $j \in \mathscr{J}$ is unfinished at time t in the schedule, there is a job $j' \in \mathscr{J}$ with $r(j') \geq r(j)$ that is active at time t.*

Proof. Assume that t^* is the smallest time-index where the theorem is not true. I.e., a job $j \in \mathscr{J}$ is unfinished, but all active jobs at time t^* have resource requirement strictly smaller than j. If there are several jobs with that property, let j be the one of largest resource requirement $r(j)$.

First of all we observe that $t^* > 0$ as the job of largest resource requirement is always scheduled at time 0. As the statement is true at time $t^* - \epsilon$ for every small enough $\epsilon > 0$, we conclude that there is a job j' with $r(j') > r(j)$ that finishes at time t^*. As no job of resource requirement at least $r(j)$ was scheduled at time t^*, we conclude that none of those jobs was available. In particular, j was not available. As the resource requirement of $r(j)$ would not lift the resource usage above one, we conclude that the resource limitation is not the reason for j

being unavailable. Hence it follows that j is not ready, i.e., a job of larger resource requirement on the same machine is still unfinished. This is a contradiction to the choice of j as the unfinished job of largest resource requirement. □

We now show the previously mentioned structural property of the generated schedule. Once the resource usage of the schedule drops below $1-\beta$, it will never rise above this threshold again.

Lemma 4.3.8. *Consider the schedule generated by the algorithm. For a time-index t, let $\mathscr{J}(t)$ denote the set of jobs that are active at time t. Let*

$$\hat{t} := \min\left\{ t \,:\, \sum_{j\in\mathscr{J}(t)} r(j) < 1-\beta \right\}.$$

Then $\sum_{j\in\mathscr{J}(t)} r(j) < 1-\beta$ for all $t \geq \hat{t}$. Moreover, at every time-index $t \geq \hat{t}$, a machine is idle only if all of its jobs are finished.

Proof. Observe that Lemma 4.3.7 implies that all jobs with $r(j) \geq \frac{1}{2}$ are finished before time $\hat{t}$: If there is such a job, then $\beta = \frac{1}{2}$. As long as there is an unfinished job of resource requirement at least $\frac{1}{2}$, there is also an active job of that resource requirement. Hence the resource usage is at least $\frac{1}{2} = 1-\beta$.

We conclude that after time $\hat{t}$, only jobs of resource requirement at most β are active. This implies that a machine k that is idle at time $\hat{t}$ has no more jobs to process: If there would be jobs left, at least one of them would be available and therefore get scheduled. As by construction of the algorithm, the resource requirement of each single machine is monotonously decreasing, we conclude that the resource usage of the schedule after time $\hat{t}$ is monotonously decreasing and the resource usage statement follows.

As the resource usage at all times $t \geq \hat{t}$ is at most $1-\beta$, with the same argument as before we see that at time t, only machines that have no more job left to process are idle. □

We are now ready to prove the following approximation guarantee for our algorithm:

Theorem 4.3.9. *Algorithm 4 is a ρ-approximation algorithm, where*

$$\rho := \frac{1}{1-\beta} + \left(1 + \frac{\tau}{OPT(\mathscr{J})}\right)$$

Proof. Let $\hat{t}$ be as in Lemma 4.3.8. The lemma states that for each $t \in [0,\hat{t})$, the resource consumption is at least $1-\beta$. On the other hand, we have seen in Corollary 4.3.6 that the total resource consumption of all jobs is at most $OPT(\mathscr{J})$. Hence $\hat{t}\cdot(1-\beta) \leq OPT(\mathscr{J})$ holds which implies

$$\hat{t} \leq \frac{1}{1-\beta} OPT(\mathscr{J}).$$

After time $\hat{t}$, Lemma 4.3.8 shows that the resource constraints are not a limitation anymore: Each processor is busy until all of its jobs are finished. Hence, an upper bound for the makespan of the schedule starting at time $\hat{t}$ is given by the processing load bound from Corollary 4.3.6: $OPT(\mathscr{J}) + \tau$. □

As $\tau \leq OPT(\mathscr{J})$ and $\beta \leq \frac{1}{2}$, we get the following corollary.

Corollary 4.3.10. *The algorithm is a* 4*-approximation algorithm.*

We conclude the section briefly discussing the case where preemptive schedules are allowed. Algorithm 4 computes a non-preemptive schedule. However, observe that for the scheduling problem with costs, it does not matter whether we allow preemptive schedules or not. The minimum makespan is the same in both cases. We conclude that the derivations in Corollary 4.3.5 and Corollary 4.3.6 hold true even if $OPT(\mathscr{J})$ denotes the optimal makespan of a preemptive schedule. We conclude that our algorithm is also an approximation algorithm for preemptive schedules with same approximation factors as stated in Theorem 4.3.9 and Corollary 4.3.10.

4.3.3 Improving the Approximation Ratio to 3.75

We now describe a few adaptations on the algorithm to improve the approximation guarantee to 3.75. As mentioned previously, unlike the 4-approximation algorithm from Section 4.3.2, this is no longer independent work as the ideas for the improvement are taken from Grigoriev et al. [GSU07]. We present it anyway for the sake of completeness. Also the analysis of *our* modified algorithm appears to be simpler than the case based analysis of Grigoriev et al. [GSU07].

To improve the approximation guarantee, both phases of our algorithm need to be modified. We first discuss the second phase. Recall that we argued that up to some time $\hat{t}$, the resource usage in the schedule is always at least $1 - \beta \geq \frac{1}{2}$. We would like to have a stronger bound than $\frac{1}{2}$. To achieve this, we change the definition of *availability* slightly. Recall that a job was available to be scheduled at a time by the list scheduler, if all jobs with larger resource requirement assigned to the same machine are finished, and scheduling the job at time t does not violate the resource constraint. For jobs whose resource requirement is less than $\frac{1}{3}$, we add a third condition: for such a job to be available at time t, all jobs of resource requirement at least $\frac{1}{3}$ must have been scheduled already at time t. We stress that they do not need to be finished already at time t, but all of them must be assigned to the schedule by that time.

The effect of this is as follows. As before, first all jobs of resource requirement of more than $\frac{1}{2}$ are scheduled sequentially. In parallel, there might be some other jobs of resource requirement at least $\frac{1}{3}$ active, but no jobs of resource requirement less than $\frac{1}{3}$ by the new definition of availability. Once all jobs of resource requirement more than $\frac{1}{2}$ are finished, and there are at least two jobs of resource requirement at least $\frac{1}{3}$ that are unfinished, at least two of them get

executed in parallel. If there is at most one unfinished job left of resource requirement at least $\frac{1}{3}$, the remaining jobs get scheduled as before in the old algorithm. In terms of resource usage, we can now show the following:

Lemma 4.3.11. *Consider a schedule generated by the modified algorithm. Let t_1 be the time where the last job of resource requirement more than $\frac{1}{2}$ finishes. Let $t_2 \geq t_1$ be the smallest time index where the resource usage drops below $\frac{2}{3}$. Then at every time-index $t \geq t_2$, a machine is idle only if all of its jobs are finished.*

Proof. By construction of the algorithm, at time t_2, there is at most one unfinished job of resource requirement at least $\frac{1}{3}$ left. Otherwise the algorithm would schedule two of those jobs in parallel, raising the resource usage above $\frac{2}{3}$. We conclude that after time t_2, the only jobs left to schedule are jobs of resource requirement at most $\beta := \frac{1}{3}$. By the lines of proof of Lemma 4.3.8, we conclude that a machine can only be idle because all of its jobs are finished. □

Observe that the benefit we get from the modified algorithm is a better bound on the resource usage: during time interval $[t_1, t_2)$, the resource usage is at least $\frac{2}{3}$. However, the resource usage in time interval $[0, t_1)$, i.e., where jobs of resource requirement of more than $\frac{1}{2}$ are running, can still be arbitrarily close to $\frac{1}{2}$. We have no control over that. This is where a modification of the first phase comes into play, more precisely a modification of the *costs*. Recall that we set the costs for the algorithm by Shmoys and Tardos (Theorem 4.3.1) to the resource consumption. The idea of Grigoriev et al. [GSU07] is to increase the weight of the jobs of resource requirement of more than $\frac{1}{2}$. If σ is the machine assignment for a solution of makespan T, then

$$\sum_{j \in \mathcal{J} \,:\, r(j,\sigma(j)) > \frac{1}{2}} p(j,\sigma(j)) \leq T,$$

as none of these jobs can run in parallel. From Lemma 4.3.4 we know that the total resource consumption also cannot exceed T:

$$\sum_{j \in \mathcal{J}} r(j,\sigma(j)) \cdot p(j,\sigma(j)) \leq T.$$

We conclude that the solution then also satisfies the following conic combination of the two inequalities from above:

$$\frac{3}{2} \sum_{j \in \mathcal{J}} r(j,\sigma(j)) \cdot p(j,\sigma(j)) + \frac{1}{4} \sum_{j \in \mathcal{J} \,:\, r(j,\sigma(j)) > \frac{1}{2}} p(j,\sigma(j)) \leq \frac{7}{4} T \tag{4.7}$$

Redefining costs based on this, we get the following lemma.

Lemma 4.3.12. *Given an instance $\mathscr{J}$, define costs*

$$c(j,k) := \begin{cases} \frac{3}{2} r(j,k) \cdot p(j,k), & \text{if } r(j,k) \leq \frac{1}{2} \\ \frac{3}{2} r(j,k) \cdot p(j,k) + \frac{1}{4} p(j,k), & \text{else} \end{cases}$$

for each job/machine pair. If there is a schedule of makespan T for the resource constrained scheduling problem, then there is a solution to the scheduling problem with costs of makespan T and cost at most $\frac{7}{4}T$.

Proof. The costs are defined in such a way that the solution with machine assignment σ from the discussion above resemble the left hand side of the valid inequality (4.7). □

The new approximation algorithm now runs the algorithm from Theorem 4.3.1 with the modified costs, then runs the modified algorithm for Phase 2. We claim that this algorithm is a 3.75-approximation algorithm.

Theorem 4.3.13. *The algorithm is a* 3.75 *approximation algorithm.*

Proof. Assuming that T is the "right" guess for the makespan, let σ be the machine assignment computed in the first phase. Let t_1 and t_2 be as in Lemma 4.3.11. Let $\ell_1 := |[0, t_1)|$ and $\ell_2 := |[t_1, t_2)|$ and $\ell := \ell_1 + \ell_2$. Also let C be the total amount of resource consumption processed by the algorithm during the time interval $[0, t_2)$. As by Lemma 4.3.11, the resource usage during interval $[0, t_1)$ is at least $\frac{1}{2}$, and during $[t_1, t_2)$ it is at least $\frac{2}{3}$ we conclude

$$C \geq \frac{1}{2}\ell_1 + \frac{2}{3}\ell_2 = \frac{2}{3}\ell - \frac{1}{6}\ell_1.$$

We reformulate this to

$$\ell \leq \frac{3}{2}C + \frac{1}{4}\ell_1.$$

With Lemma 4.3.12 we know that Equation (4.7) is satisfied. Note that $C = \sum_{j \in \mathscr{J}} r(j,\sigma(j)) \cdot p(j,\sigma(j))$ and $\ell_1 = \sum_{j \in \mathscr{J} \,:\, r(j,\sigma(j)) > \frac{1}{2}} p(j,\sigma(j))$. We conclude that

$$\ell \leq \frac{7}{4}T.$$

The rest of the proof is by the lines of Theorem 4.3.9. After time t_2, Lemma 4.3.11 shows that the resource constraints are not a limitation anymore: Each processor is busy until all of its jobs are finished. Hence, an upper bound for the makespan of the schedule starting at time t_2 is given by the processing load bound from Corollary 4.3.6, i.e., $2 \cdot T$.

We conclude that the total makespan of the algorithm is at most $\ell + 2 \cdot T = 3.75 \cdot T$. □

4.4 Further Research Questions

For the machine model of identical machines, we have matching approximation and inapproximability results for most cases. Only for the variant of finding non-preemptive schedules where we have no restriction on the number of machines, a small gap remains. Here we know that it is NP-hard to approximate the problem with a factor between than $\frac{3}{2}$. Our algorithm on the other hand only achieves an approximation guarantee of 2 (asymptotically). As this variant probably is the most interesting one, it would be very desirable to close this gap. Moreover, the approximation scheme is a purely theoretical construction, and totally insufficient for any practical purposes. It would be very interesting to see an efficient combinatorial algorithm that beats Garey and Grahams classic list scheduler [GG75].

Unlike in the identical machine model, the gap between best known inapproximability bound and best approximation algorithm is quite large, i.e., $\frac{3}{2}$ vs. 3.75. For that reason it would be desirable to narrow this gap, either by proving better hardness bounds or by finding better approximation algorithms. Also, as the approximation algorithm produces non-preemptive schedules and works for an arbitrary number of machines, it might be worth checking whether the approximation guarantee can be improved for the other cases exploiting the fact that preemption is allowed or the number of machines is bounded by a constant.

In related work, the problem is often considered in a more general setting where there are s independent resources that are used by the jobs, and the schedule must make sure that for none of the resources, the respective resource requirements of active jobs exceed the limits. This rises the question whether our $(2+\epsilon)$-approximation scheme can be generalized to this setting as well. If s is upper bounded by a constant, some of the sub-components of the algorithm generalize nicely, while others do not. Already looking at the natural generalization of the analysis of the greedy list scheduler when dealing with tiny/thin jobs however suggests that the best we can hope for following this approach is a $(s+1+\epsilon)$-approximation scheme. Considering that Garey and Graham have a $s+2$, approximation algorithm, the difference gets negligible for larger s. Thus it is probably not worth the effort to study this generalization of the algorithm, but rather new concepts are needed. Note that a special case of our problem with s resources is the s-dimensional vector bin packing problem (just like BIN-PACKING is a special case of the (standard) resource constrained scheduling problem). If s is bounded by a constant, there is a $O(\log(s))$-approximation algorithm by Chekuri and Khanna [CK04]. Bansal et al. [BCS06, BCS09] give a randomized approximation algorithm with approximation ratio arbitrarily close to $\ln(s)+1$. Already for the case that $s=2$, Woeginger [Woe97] shows that there is no asymptotic PTAS for s-dimensional BIN-PACKING unless $P=NP$. If s is not bounded by a constant, the problem is NP-hard to approximate within a factor of $s^{\frac{1}{2}-\epsilon}$ [CK04]. So something one could aim for is a $\log(s)$-approximation for our problem with s resources, where s is a constant.

5 Resource Constrained Scheduling: Computing Schedules Online

5.1 Introduction

In Chapter 4 we discussed approximation algorithms that compute offline schedules for the resource constrained scheduling problem. The approximation algorithms had full knowledge of the instance $\mathcal{J}$ when computing a schedule. This assumption however is not always realistic. For example, computation servers might run for a long time, even years. By the time these machines are turned on, it is not known yet which jobs they will have to execute during their lifetime. Instead the system learns about them over time and needs to adapt to them online, still making efficient use of the resources available without being able to change scheduling decisions of "the past". Algorithms to compute schedules under these restrictions are called *online algorithms*, and this chapter is devoted to the study of these.

Online algorithms are particularly interesting as they have immediate consequences for certain real-time scheduling problems, where tasks periodically generate jobs that need to be finished within a certain amount of time. This is because every online algorithm can be used also as a scheduler for the real-time task systems. Before we discuss any more details, we introduce our models of study more formally.

5.1.1 The Model

We assume that the reader is familiar with concepts and notation from Section 4.1.1. As in Chapter 4, an instance is given by a set of jobs $\mathcal{J}$ and m machines, where each job $j \in \mathcal{J}$ has a processing time $p(j)$ and a resource requirement $r(j)$. Additionally, every job now also comes with a *release date* $rel(j)$ and a *deadline* $d(j)$. The goal is to find a feasible schedule for the jobs $\mathcal{J}$ on m machines, with the additional constraint that every job $j \in \mathcal{J}$ is executed only during the time interval $[rel(j), d(j))$. We assume that all release dates and deadlines are integer.

As the problem of computing these schedules is NP-hard, we again resort to the study of

approximation algorithms. To to so, we need the concept of *machine speedup*. A machine of *speed* α can process jobs α times as fast, meaning that in a feasible schedule, it is sufficient to assign exclusive running time of $\frac{p(j)}{\alpha}$ units instead of $p(j)$ to each job j. For some $\alpha \geq 1$ we say that an algorithm is an α-approximation algorithm, if for every instance $\mathcal{J}$ that has a feasible solution on m speed-one machines, it finds a solution on m speed-α machines.

As mentioned previously, in this chapter we focus on the study of online algorithms. This is modeled as follows. In contrast to the offline case, an online algorithm learns about the existence of a job $j \in \mathcal{J}$ only at its release date $rel(j)$. It is allowed to modify the schedule when it learns about a new job. However, only modifications to time-slots after time $rel(j)$ are allowed. Albeit having less information about the instance than offline algorithms, when assessing the performance of an online algorithm, it is customary to compare to optimal offline schedules. An online algorithm is α*-competitive*, if for every instance where there is a feasible (offline) schedule on m speed-one machines, it finds a feasible online schedule on speed α machines.

Unlike the previous chapter, the standard model we consider here allows preemptive and migratory schedules, i.e., the execution of jobs can be interrupted and continued on other machines. The reason is that the non-preemptive case is "not interesting" in terms of approximation algorithms: already in the absence of the resource constraint it is folklore that there is no algorithm of bounded competitive ratio for non-preemptive schedules.

The real-time scheduling problem that we study here is different from the periodic maintenance problem discussed in Chapter 2 and Chapter 3. As in the periodic maintenance problem, an instance consists of a set of tasks $\mathcal{T}$ and m machines. Now each task $\tau \in \mathcal{T}$ is characterized by an execution time $c(\tau)$, a minimum inter-arrival separation $s(\tau)$ and a (relative) deadline $d(\tau)$. Each task releases jobs over time, possibly infinitely many. However the interval between the release of two jobs from the same task has length at least $s(\tau)$. Each job released by task τ has an execution time of $c(\tau)$ and a deadline that is $d(\tau)$ time units after its release time. This model is called the *sporadic task model*. The goal is to have an algorithm that adaptively computes a feasible schedule on speed-α machines, provided that a schedule on speed-one machines exists, where α should be as small as possible. As in the online scheduling model, preemption and migration is allowed. In real-time scheduling literature, such an algorithm is usually referred to as an algorithm that is *optimal* on speed-α machines. For a more extensive introduction into real-time scheduling problems, we refer to part four of a book by Leung et al. [LKA04].

We consider the *resource constrained real-time scheduling* problem where every task τ also has a resource requirement $r(\tau)$. A job released by τ inherits this resource requirement, and a feasible schedule has to obey the resource constraint, enforcing that the resource usage of simultaneously active jobs never exceeds 1.

Observe that real-time scheduling algorithms are somewhere in between offline and online

algorithms. On the one hand, they know the full real-time scheduling instance in advance. In particular they know which types of jobs to expect. On the other hand however, they do not know in advance when exactly the jobs are released, but rather learn about them online. Clearly, every online algorithm can be used also as a real-time scheduling algorithm, by simply ignoring the additional knowledge about the task system. In particular, an α-competitive online algorithm is a real-time scheduling algorithm that is optimal on speed-α machines. The converse of course is not always true.

5.1.2 Related Work

To the best knowledge of the author, there is no previous work regarding online algorithms for the resource constrained scheduling problem. The basic problem, i.e., the online problem ignoring the resource constraint, however has been studied intensively and still is a very active research topic. A good overview of the "state of the art" is given in a recent survey paper by Davis and Burns [DB11]. Here we only discuss related literature that is most relevant to this work. The offline problem can be solved optimally in polynomial time, for example via a linear programming formulation or even combinatorially by solving a maximum flow problem [Der74]. (We stress again that we consider preemptive and migratory schedules.) For online algorithms however it was shown by Dertouzos and Mok [DM89] that there is no 1-competitive online algorithm, when studying instances with more than one machine.

Two popular online algorithms are *earliest deadline first* (EDF) [Der74] and *least laxity first* (LLF) [Mok76]. EDF schedules jobs prioritized by their deadline, jobs with earliest deadline having best priority. In LLF, jobs are prioritized by their laxity, i.e., the difference of deadline and remaining processing time, jobs with smallest laxity having best priority. At any time t, the (at most) m unfinished jobs with best priority are scheduled to the m machines. Phillips et al. have shown that both EDF and LLF are $(2-\frac{1}{m})$-competitive [PSTW97]. In particular they are optimal for the single machine case, but asymptotically approach a competitive ratio of 2 for larger number of machines. They also showed that every online algorithm has a competitive ratio of at least $\frac{6}{5}$ [PSTW97].

Lam and To [LT99] marginally improve lower and upper bounds by providing an online algorithm that is $(2-\frac{2}{m+1})$-competitive, and showing that every online algorithm has a competitive ratio of at least $\frac{1}{2\sqrt{2}-2} \approx 1.207 > \frac{6}{5}$. Still, their algorithm has an asymptotic competitive ratio of 2 just like EDF and LLF. Their algorithm is rather complex, however the basic idea is to first compute an infeasible online schedule which they named the *yardstick* schedule. Based on this, they then compute a feasible online schedule. The yardstick schedule is infeasible because it allows jobs to be executed in parallel on several machines. This parallelization is allowed however only under certain conditions. A nice property of the yardstick schedule is that it is an online schedule where all jobs are processed within their release date/deadline interval, even on speed-one machines. Recently, this two step idea of first computing a yardstick schedule, and then converting it to a feasible schedule was picked up by Anand, Garg and

Megow [AGM11] to design an α_m-competitive algorithm, where $\alpha_m := \frac{1}{1-(1-\frac{1}{m})^m}$. This number asymptotically converges to $e/(e-1) \approx 1.56$.

EDF, the algorithm by Lam and To, and the algorithm by Anand, Garg and Megow have one thing in common: Their scheduling decisions depend on the relative order of the deadlines, but not on their actual values. Lam and To [LT99] have shown that these kinds of algorithms have a competitive ratio of at least α_m, which shows that the online algorithm by Anand, Garg and Megow is the best possible algorithm of this kind.

The situation for the resource constrained real-time scheduling is similar. There are no previous results regarding *this* problem, but the basic problem of real-time scheduling sporadic task systems is widely studied and still a very active topic of research. As mentioned previously, when the system has only one machine, EDF is 1-competitive, which translates to an optimal real-time scheduling algorithm: if there is a feasible schedule on a single machine, EDF will compute it. As the schedule however is of infinite length, this does not automatically imply that we can *test* whether a feasible schedule exists. In fact, it has been shown recently by Eisenbrand and Rothvoß [ER10] that testing feasibility is coNP-hard. A short no-certificate however exists for the single-machine case as shown by Baruah et al. [BRH90] which implies that the problem is coNP-complete. In the multi-machine case, for a long time it was not even known how to test feasibility of an algorithm, or the existence of a feasible schedule at all (no matter which running time of the test). Recently Bonifaci and Marchetti-Spaccamela [BMS10] found a feasibility test whose running time however is exponential in the input size. Approximate feasibility tests (in the sense that they either certify feasibility on speed-α machines or infeasibility on speed one machines) exist however. Baruah and Baker [BB08] give an $\frac{2}{3-\sqrt{5}} \approx 2.61$ feasibility test. This was later improved by Bonifaci et al. [BMSS08] to a $(2-\frac{1}{m}+\epsilon)$-approximation scheme for the feasibility test. Our focus in this work is more on scheduling algorithms and less on feasibility tests. The best known real-time scheduling algorithm for the problem is the $e/(e-1)$-competitive online algorithm by Anand, Garg and Megow [AGM11] mentioned earlier. It is notable that the best real-time scheduling algorithm known so far is actually an online algorithm, i.e., it does not make use of any information about the real-time scheduling instance.

5.1.3 Our Contribution and Outline

We consider the resource constrained scheduling problem in the online setting and provide an online algorithm that is $(5+e/(e-1))$-competitive, which is roughly 6.582. To the best knowledge of the author, this is the first online algorithm for the resource constrained scheduling problem. This also translates to a real-time scheduling algorithm that is optimal on speed-$(5+e/(e-1))$ machines. The algorithm is presented in Section 5.2.

We also show that every online algorithm for the resource constrained scheduling problem has a competitive ratio of at least $\frac{4}{3}$, thus improving on the best known lower bound for the

basic problem of 1.207 by Lam and To [LT99]. The bound is established in Section 5.3

Both results are proved in a rather straightforward way. They should be seen as preliminary results it should not be too difficult to improve on. At the end of the chapter in Section 5.4 we discuss ideas for improvements and further interesting research questions.

5.2 An Online Algorithm for Resource Constrained Scheduling

We now present an online algorithm for preemptive resource constrained scheduling on identical machines which is α-competitive, where $\alpha := 5 + e/(e-1) \approx 6.582$. The main idea of the algorithm is as follows. Jobs are classified by their resource requirement, effectively partitioning an instance into several sub-instances. Making use of preemption, these sub-instances can be treated independently. Given separate schedules for each of the sub-instances, they are merged into one schedule by executing them alternating on faster machines. This idea is formalized in the following statement.

Lemma 5.2.1. *Given an online algorithm A_1 that is α_1-competitive for some family of instances, and an online algorithm A_2 that is α_2-competitive for another family of instances. There is an online algorithm that is $(\alpha_1 + \alpha_2)$-competitive for any instance that can be written as the union of instances from the two respective families.*

Proof. Consider a feasible instance $\mathscr{J} = \mathscr{J}_1 \dot\cup \mathscr{J}_2$, where $\mathscr{J}_1$ and $\mathscr{J}_2$ are instances of the two families from the statement of the lemma. Hence A_1 computes an online schedule for $\mathscr{J}_1$ on speed-α_1 machines, and A_2 computes an online schedule for $\mathscr{J}_2$ on speed-α_2 machines, The online algorithm for $\mathscr{J}$ now works as follows: For each unit-length time slot $[t, t+1)$, it uses A_1 and A_2 to compute their respective schedules for that time slot. It then scales the schedule from A_1 by $\alpha_1/(\alpha_1 + \alpha_2)$ and the schedule from A_2 by $\alpha_2/(\alpha_1 + \alpha_2)$. The new schedule for that time slot now runs the scaled schedule from A_2 in interval $[t, t + \alpha_1/(\alpha_1 + \alpha_2))$, and the scaled schedule from A_2 in the remaining interval. Note that due to the fact that the machines the new schedule is executed on have speed $\alpha_1 + \alpha_2$, exactly the same amount of processing is done for instance $\mathscr{J}_1$ and instance $\mathscr{J}_2$ respectively as in the old schedules during the interval $[t, t+1]$. Since by assumption, all deadlines are integers and in the schedule from A_1 and A_2 are feasible and all jobs meet their deadline, the same is true for the new schedule. □

As mentioned previously, we will classify jobs by resource requirement. We have three classes and call jobs *slim*, *medium* or *thick* depending on their resource requirement being at most $\frac{1}{m}$, between $\frac{1}{m}$ and $\frac{1}{2}$, or more than $\frac{1}{2}$ respectively. Following that classification, we partition an instance $\mathscr{J}$ into three groups

$$SJ := \left\{ j \in \mathscr{J} \,:\, r(j) \leq \frac{1}{m} \right\},\ MJ := \left\{ j \in \mathscr{J} \,:\, \frac{1}{m} < r(j) \leq \frac{1}{2} \right\} \text{ and } TJ := \left\{ j \in \mathscr{J} \,:\, r(j) > \frac{1}{2} \right\}.$$

Each of these groups comes from different families of jobs that can be treated more easily. For

each of them, we will next specify an online algorithm and give bounds on their competitive ratios. Using Lemma 5.2.1 we can then combine them to get an online algorithm for the full instance with the claimed competitive ratio.

Family of slim jobs

For the slim jobs, we observe that any m of them can be executed simultaneously without violating the resource constraint. This is by construction of SJ, as the resource requirement of each job is at most $\frac{1}{m}$. Hence, we can use any known (online) algorithm for the basic problem to schedule the jobs. As previously mentioned, the currently best known online algorithm by Anand, Garg and Megow [AGM11] has a competitive ratio of $e/(e-1)$.

Family of medium jobs

Our aim here is to have an online algorithm that is 4-competitive for the family of medium jobs. To this end, we first give an online algorithm that computes a feasible schedule on machines with speed 2 and a total resource limit of $\frac{3}{2}$ instead of 1. Afterwards we show how to turn this into a feasible schedule on speed-4, resource-1 machines. The group MJ is defined in such a way that essentially we can forget about the limited number of machines. Observe that as the resource requirement of each job is more than $\frac{1}{m}$, we can conclude that every set of jobs from MJ whose sum of resource requirement is at most 1 can have at most $m-1$ elements, thus they can always run simultaneously and even keep at least one machine idle. This will be important later in the analysis of the algorithm that we specify next.

The jobs are scheduled using an earliest deadline first algorithm which is defined as follows. At each time index $t \in \mathbb{N}_0$, the algorithm decides which jobs to schedule during time-slot $[t, t+1)$. It considers all jobs that are known and unfinished at time t. The jobs are sorted by their deadline, earliest deadline first. Now, while there is a free machine, the algorithm takes the first job from the sorted list whose resource requirement does not exceed the available remaining resource, and assigns it to a machine. We claim that, if there is a feasible schedule for instance MJ on m speed-1 and resource-1 machines, then this EDF-type algorithm produces a feasible schedule on m speed-2 and resource-$\frac{3}{2}$ machines. The analysis of the algorithm is essentially by the lines of the analysis of Phillips et al. [PSTW97] which shows that EDF in the basic setting is $(2-\frac{1}{m})$-competitive. To show our claim, we start with some trivial observations:

Lemma 5.2.2. *Consider a schedule generated by the algorithm. If all machines are busy at time t, then the resource usage at time t is at least* 1.

Proof. As every job of group MJ has resource requirement at least $\frac{1}{m}$, the aggregated resource requirement of m jobs is always at least one. □

For a job j, let $P(j,t)$ be the amount of processing of job j done by our algorithm till time t

in our schedule. For a job set $\mathscr{J}$, let $P(\mathscr{J}, t) := \sum_{j \in \mathscr{J}} P(j, t)$. Similarly, let $M(j, t)$ be the total amount of resource consumption of job j already processed by the schedule up to time t, i.e., $M(j, t) := r(j) \cdot P(j, t)$ and $M(\mathscr{J}, t) := \sum_{j \in \mathscr{J}} M(j, t)$.

Now consider a feasible schedule on speed-1, resource-1 machines. We define $P'(j, t)$, $M'(j, t)$, etc. as the processing time and resource consumption already processed up to time t in *that* schedule. We claim that at any time, our algorithm outperforms this schedule in terms of processed resource consumption:

Lemma 5.2.3. *For each time t we have $M(MJ, t) \geq M'(MJ, t)$.*

Proof. Assume that the claim does not hold. Let t be minimal with the property that $M(MJ, t) < M'(MJ, t)$. As we have $M(MJ, t) < M'(MJ, t)$, there must be a job $j \in MJ$ with $M(j, t) < M'(j, t)$. Consider the time interval $I := [rel(j), t]$. By minimality of t we know that $M(MJ, rel(j)) \geq M'(MJ, rel(j))$ and therefore

$$M(MJ, t) - M(MJ, rel(j)) < M'(MJ, t) - M'(MJ, rel(j)) \tag{5.1}$$

Let x be the amount of time in interval I where the resource used by our algorithm is at least 1. Let y be the amount of time where the resource used is less than 1. Clearly $x + y = t - rel(j)$. At time-indexes t where the resource usage is less than 1, observe that job j must be active. Assume otherwise. Then as observed in the beginning of this section, as the resource usage is less than 1, at most $m - 1$ jobs are active as every job from MJ has resource requirement more than $\frac{1}{m}$. Also the available resource (on our resource $\frac{3}{2}$ machines) is at least $\frac{1}{2}$, whereas $r(j) \leq \frac{1}{2}$. Finally, job j is not finished at time t. Hence the algorithm can schedule job j at time t and therefore does so.

As j is active whenever the resource usage is below 1, we conclude that the resource usage is at least $r(j)$. Hence we can lower bound the total resource consumption processed by our algorithm during the interval $[rel(j), t)$ as follows

$$M(MJ, t) - M(MJ, rel(j)) \geq 2 \cdot (x + y \cdot r(j)).$$

The factor of 2 comes from the fact that our machines have speed 2.

On the other hand, the optimal schedule running on speed-1 and resource-1 machines can process at most $(t - rel(j))$ units of resource consumption during that time interval

$$M'(MJ, t) - M'(MJ, rel(j) \leq t - r(j).$$

With (5.1) we conclude that

$$2 \cdot (x + y \cdot r(j)) < t - rel(j). \tag{5.2}$$

As $M(j,t) < M'(j,t)$, we conclude that algorithm A' processed more resource of job j than algorithm A did. Hence

$$2y < t - rel(j).$$

Adding this with a factor of $(1 - r(j))$ to inequality (5.2) we get

$$2(x + y) < (2 - r(j))(t - rel(j)).$$

As $(2 - r(j)) < 2$, we get that $x + y < t - rel(j)$. This is a contradiction to the definition of x and y. □

Now we can show the desired guarantee of the algorithm in terms of machine speedup and resource augmentation.

Lemma 5.2.4. *If there is a feasible schedule for MJ on m machines of speed 1 and resource 1, then our algorithm feasibly schedules the instance on m machines with speed 2 and resource 3/2.*

Proof. Let $1,\dots,n$ be the jobs of the instance, sorted by their deadline. The proof is by induction on $j = 1,\dots,n$. The induction hypothesis is that the algorithm schedules jobs $1,\dots,j$ feasibly. Clearly, for $j = 1$ the algorithm works. Now assume that jobs $1,\dots j$ meet their deadline. We want to show that job $j+1$ also meets its deadline. Observe that we can ignore jobs $j+2,\dots,n$ as they have no influence on the schedule for the other jobs in our schedule. Observe that by induction hypothesis, at time $d(j+1)$ all jobs $1,\dots,j$ are finished. Hence we have $M(\{1,\dots,j\},d(j+1)) = M'(\{1,\dots,j\},d(j+1))$. By Lemma 5.2.3 we have that $M(\{1,\dots,j,j+1\},d(j+1)) \geq M'(\{1,\dots,j,j+1\},d(j+1))$ We conclude that $M(j+1,d(j+1)) \geq M'(j+1,d(j+1)) = p(j+1)$ which shows that job $j+1$ is finished at time $d(j+1)$ as well. □

It remains to turn this algorithm into a 4-competitive algorithm (for resource 1 machines).

Lemma 5.2.5. *There is an online algorithm that feasibly schedules the instance MJ on m machines of speed 4, provided that there is a feasible schedule for MJ on m machines of speed 1.*

Proof. Consider the schedule from Lemma 5.2.4. By the definition of the EDF type algorithm from above, jobs are only preempted at integral time-steps. For each time-step $[t,t+1)$ denote by $\mathcal{J}_t$ the jobs which are executed in that time-step. As the algorithm respects the resource constraint, it holds that $\sum_{j\in\mathcal{J}_t} r(j) \leq 3/2$. Also, $r(j) \leq 1/2$ for each job $j \in MJ$. We claim that we can partition $\mathcal{J}_t$ into two sets $\mathcal{J}_t^{(1)}$ and $\mathcal{J}_t^{(2)}$ such that $\sum_{j\in\mathcal{J}_t^{(1)}} r(j) \leq 1$ and $\sum_{j\in\mathcal{J}_t^{(2)}} r(j) \leq 1$. This can be achieved in a greedy manner: consider the jobs in an arbitrary order and assign them greedily to one of the two sets such that the total resource requirement of no set exceeds 1. Using similar reasoning as in Lemma 5.2.1, our new scheduler for speed-4 machines simply executes the jobs $\mathcal{J}_t^{(1)}$ in the first half of interval $[t,t+1)$ and $\mathcal{J}_t^{(2)}$ in the second half. Analogous

to the proof of Lemma 5.2.1 it is straightforward to see that every job gets the same amount of processing in interval $[t, t+1)$ with the new schedule on speed-4 machines as in the old schedule on speed-2 machines. □

Family of thick jobs

Observe that at most one thick job can run at a time without violating the resource constraint. Hence, if there is a feasible schedule for TJ on m speed one machines, then there is a feasible schedule on one speed-one machine. It is known that on one machine the EDF-schedule meets all deadlines if such a schedule exists, see e.g., [DM89] or [PSTW97]. Hence, there is an online algorithm, namely EDF, which computes a feasible schedule on machines with speed 1 (if the instance is feasible).

The full algorithm

As seen above, the jobs in SJ can be scheduled online using m machines of speed $e/(e-1)$, the jobs in MJ can scheduled online using m machines of speed 4, and the jobs in LJ can be scheduled using one machine with speed 1, assuming that the input instance is feasible. Applying Lemma 5.2.1 twice yields the following theorem.

Theorem 5.2.6. *There is a $5+e/(e-1)$-competitive online algorithm for the resource constrained scheduling problem.*

5.3 Lower Bounds on the Competitive Ratio

Lower bounds on the competitive ratio are usually established using an adversary model, where the adversary can choose the instance depending on the scheduling decisions of an online algorithm. To prove a lower bound of α on the competitive ratio, we need to describe an adversary that adaptively constructs an instance which, no matter what kind of schedule the online algorithm produces, leads to infeasibility on machines with speed at most α. We show that every online algorithm for the resource constrained scheduling problem has a competitive ratio of at least $\frac{4}{3}$, which is slightly more than the best known lower bound for the basic setting of $\frac{1}{2\sqrt{2}-2} \approx 1.207$ from [LT99].

Theorem 5.3.1. *The competitive ratio of every online algorithm for the resource constrained scheduling problem is at least $\frac{4}{3}$.*

Proof. Consider the case of three machines of speed α, i.e., $m = 3$. We want to find a lower bound on α. At time 0, three jobs 1, 2 and 3 are released with the following parameters $p(1) = p(2) = p(3) = 1$, $rel(1) = rel(2) = rel(3) = 0$, $d(1) = d(2) = d(3) = 2$, $r(1) = r(2) = 0.1$, $r(3) = 1$.

At a time $t \in [0,1)$, a scheduler can either run job 3, or jobs 1 and 2 in parallel. Assume that

at time 1, the scheduler has processed a units of jobs 1 and 2, and b units of job 3. Then $a+b=\alpha$. The adversary now adds more jobs, depending on whether a or b is larger. If $a \geq \frac{\alpha}{2}$, the adversary releases a fourth job at time 1 with parameters $p(4)=1$, $r(4)=0.1$, $rel(4)=1$, $d(4)=2$. Job 3 requires a remaining processing time of at least $1-\frac{\alpha}{2}$ in the time interval $[1,2)$, and it cannot be processed in parallel with job 4 because of the resource requirement. So in order to meet all deadlines, the total processing that needs to be done sequentially in the interval $[1,2)$ is $2-\frac{\alpha}{2}$, i.e., we require $\alpha \geq 2-\frac{\alpha}{2}$ which is equivalent to $\alpha \geq \frac{4}{3}$. On the other hand, a feasible offline schedule on speed-one machines would be to run job 3 during time interval $[0,1)$, and jobs $1,2,4$ during time-interval $[1,2)$.

For the second case that $b \geq \frac{\alpha}{2}$, the adversary releases two jobs 4 and 5 with parameters $p(4)=p(5)=1$, $r(4)=r(5)=0$, $rel(4)=rel(5)=1$, $d(4)=d(5)=2$. As there are only three machines available, jobs $1,2,4,5$ cannot run in parallel. The remaining processing time of jobs 1 and 2 is at least $1-\frac{\alpha}{2}$, hence the total processing that needs to be done sequentially in interval $[1,2)$ is $2-\frac{\alpha}{2}$, so analogous to the previous case we get $\alpha \geq \frac{4}{3}$. Also here a feasible offline schedule on speed-one machines exist by scheduling jobs 1 and 2 during interval $[0,1)$, and jobs 3, 4 and 5 during interval $[1,2)$. □

5.4 Further Research Questions

As the results presented in this section are quite straightforward to establish, they should be seen as preliminary results that we want to improve on. It might be possible to improve the competitive ratio of the online algorithm to $1+3\cdot\frac{e}{e-1} \approx 5.74593012$ by treating the family of medium jobs in a different way. We used EDF and showed with a modified analysis that we get a feasible schedule on speed-2 memory-$\frac{3}{2}$ machines. It should be possible to use the algorithm by Anand, Garg and Megow [AGM11] instead, but adapting the analysis proves to be more difficult. Using this algorithm would give the suggested competitive ratio. However, to significantly improve the performance ratio, we believe that one needs to get around the partitioning of the instance into subproblems, but rather deal with the full problem directly.

Considering that there is a huge gap between the competitive ratio of our algorithm and the lower bound of $\frac{4}{3}$, and the instance that we use to obtain the bound is rather simple compared to [PSTW97], it is likely that there is room for improvement as well.

Finally, it would be nice to have approximate feasibility tests as in the basic setting. This question has not been studied at all so far.

Another open problem that came up recently in the real-time scheduling community is, in a way, dual to ours. Given a sporadic real-time task system with resource requirements and an unlimited number of machines, find a schedule on speed one machines so that the peak resource usage of the schedule is minimized.

6 Summary and Conclusion

In this thesis we studied several scheduling problems that arise in modern multi-processor computer architectures. These ranged from real-time scheduling problems coming to use in safety-critical applications for the avionics industry (Chapter 2 and Chapter 3) to makespan minimization scheduling with shared resources to use modern multi-core computer architectures efficiently (Chapter 4). Finally we came back to real-time scheduling problems via considering scheduling with shared resources in an online setting (Chapter 5).

For all these problems and variants, we assessed their computational complexity in a mathematically rigorous way, both by designing approximation algorithms and by proving inapproximability results. For some of the problems, this leads to a quite precise characterization of their complexity landscape, e.g., for the periodic maintenance problem, see Table 2.1. For other problems, e.g., the resource constrained scheduling problem, we have a characterization (Table 4.1), which however is not yet complete: for some variants, we could narrow the gaps between best known approximation algorithms and inapproximability results, but not quite close it. Finally there are problems where we only scratched the surface, leaving a huge gap between lower and upper bounds, like in the case of online resource constrained scheduling discussed in Chapter 5. The two most notable results of this thesis are the following. First, there is the 2-approximation algorithm for the periodic maintenance problem with harmonic periods (Section 2.3). It is a simple combinatorial algorithm that is also efficient for practical purposes, with an elegant analysis. Also it is a tight result in the sense that there is no polynomial time algorithm with an approximation guarantee better than 2, unless $P = NP$. The other notable result is the $(2+\epsilon)$-approximation scheme for the resource constrained scheduling problem (Section 4.2.2). Although the algorithm is rather complicated and the analysis is quite involved, after almost four decades it is the first algorithm to improve on a classic result by Garey and Graham from 1975. Also the algorithms narrows the gap to the $\frac{3}{2}$-inapproximability significantly.

Next to the rather theoretic analysis of approximation algorithms, we also studied consequences for practical applications arising from our theoretical work. For the periodic maintenance problem, we not only provided a very efficient (in the practical sense) heuristic to obtain

a 2-approximation. The theoretical insights gained during the analysis yielded structural results that allow for fundamentally new integer programming formulations. Commercially available IP-solvers like CPLEX where able to deal with the new formulations much faster, rendering industrial-size instances computationally solvable to optimality for the first time. Many other algorithms presented in this work could also be applied in real-world settings as they are also "practically efficient", e.g., the online algorithm for the resource constrained scheduling problem discussed in Section 5.2. Some algorithms however, like the $(2+\epsilon)$-approximation scheme for the offline setting, are purely theoretical constructs and not applicable in practice at all.

In summary, in this thesis we were able to answer many questions regarding the problems of study, obtaining a deeper understanding of the nature of these problems. Yet, some questions remain open, and new questions arose during the study of the problems. This leaves space for further research.

Bibliography

[ADN10] C. Artigues, S. Demassey, and E. Néron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications.* ISTE, 2010.

[AGM11] S. Anand, N. Garg, and N. Megow. Meeting deadlines: How much speed suffices? In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 6755 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin / Heidelberg, 2011.

[ASBH10] A. Al Sheikh, O. Brun, and P.-E. Hladik. Partition Scheduling on an IMA Platform with Strict Periodicity and Communication Delays. In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, pages 179–188, 2010.

[ASBHP11] A. Al Sheikh, O. Brun, P.E. Hladik, and B.J. Prabhu. A best-response algorithm for multiprocessor periodic scheduling. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 228 –237, 2011.

[BB08] S. Baruah and T. Baker. Schedulability analysis of global EDF. *Real-Time Systems*, 38:223–235, 2008.

[BBNS02] A. Bar-Noy, R. Bhatia, J. Naor, and B. Schieber. Minimizing service and operation costs of periodic scheduling. *Math. Oper. Res.*, 27(3), 2002.

[BCS06] N. Bansal, A. Caprara, and M. Sviridenko. Improved approximation algorithms for multidimensional bin packing problems. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 697 –708, 2006.

[BCS09] Nikhil B., Alberto C., and Maxim S. A new approximation method for set covering problems, with applications to multidimensional bin packing. *SIAM J. Comput.*, 39(4):1256–1278, 2009.

[Bha98] R. Bhatia. *Approximation Algorithms for Scheduling Problems.* PhD thesis, University of Maryland, 1998.

[BLK83] J. Blazewicz, J.K. Lenstra, and A.H.G.Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11 – 24, 1983.

[BMS10] V. Bonifaci and A. Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. In *ESA 2010*, volume 6347 of *Lecture Notes in Computer Science*, pages 230–241. Springer Berlin / Heidelberg, 2010.

[BMSS08] V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. A constant-approximate feasibility test for multiprocessor real-time scheduling. In Dan Halperin and Kurt Mehlhorn, editors, *Algorithms - ESA 2008*, volume 5193 of *Lecture Notes in Computer Science*, pages 210–221. Springer Berlin / Heidelberg, 2008.

[BNW11] S. Baruah, M. Niemeier, and A. Wiese. Partitioned real-time scheduling on heterogeneous shared-memory multiprocessors. In *23rd Euromicro Conference on Real-Time Systems (ECRTS2011)*, 2011.

[BRH90] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990. 10.1007/BF01995675.

[BRTV90] S. Baruah, L. Rousier, I. Tulchinsky, and D. Varvel. The complexity of periodic maintenance. *Proceedings of the International Computer Symposium*, 1990.

[BT97] D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1st edition, 1997.

[CK04] C. Chekuri and S. Khanna. On multidimensional packing problems. *SIAM Journal on Computing*, page 33:837–851, 2004.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2 edition, 2001.

[CPPT07] T. G. Crainic, G. Perboli, M. Pezzuto, and R. Tadei. New bin packing fast lower bounds. *Computers & Operations Research*, 34:3439–3457, 2007.

[DB11] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, 2011.

[Der74] M.L. Dertouzos. Control robotics: The procedural control of physical processes. *IFIP Congress*, pages 807 –813, 1974.

[DM89] M.L. Dertouzos and A.K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497 –1506, 1989.

[EHN+10] F. Eisenbrand, N. Hähnle, M. Niemeier, M. Skutella, J. Verschae, and A. Wiese. Scheduling periodic tasks in a hard real-time environment. In *37th International Colloquium on Automata, Languages and Programming (ICALP2010)*, volume 37 of *Lecture Notes in Computer Science*, pages 299–311. Springer-Verlag, 2010.

[EKM+10] F. Eisenbrand, K. Kesavan, R. Mattikalli, M. Niemeier, A. Nordsieck, M. Skutella, J. Verschae, and A. Wiese. Solving an Avionics Real-Time Scheduling Problem by Advanced IP-Methods. In *18th Annual European Symposium on Algorithms (ESA2010)*, Lecture Notes in Computer Science. Springer-Verlag New York, 2010.

[ER10] F. Eisenbrand and T. Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *ACM-SIAM Symposium on Discrete Algorithms (SODA10)*, 2010.

[FdlVL81] W. Fernandez de la Vega and G. S. Lueker. Bin packing can be solved within $1+\varepsilon$ in linear time. *Combinatorica*, 1:349–355, 1981.

[FH02] K. Fleszar and K. S. Hindi. New heuristics for one-dimensional bin-packing. *Computers & Operations Research*, 29:821–839, 2002.

[GG75] M. R. Garey and R. L. Grahams. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4:187–200, 1975.

[GJ75] M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 1975.

[GJ78] M. R. Garey and D. S. Johnson. “ Strong ” NP-Completeness Results: Motivation, Examples, and Implications. *J. ACM*, 25(3):499–508, 1978.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freemann, 1979.

[Gra66] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:263–269, 1969.

[GSU07] A. Grigoriev, M. Sviridenko, and M. Uetz. Machine scheduling with resource dependent processing times. *Mathematical Programming*, 110:209–228, 2007.

[HB10] S. Hartmann and D. Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207:1–14, 2010.

[Hoc96] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* Thomson, 1996.

[HS76] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23(2):317–327, 1976.

[HS87] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[JJ98] G. A. Jones and J. M. Jones. *Elementary Number Theory*. Springer, July 1998.

[JP06] K. Jansen and L. Porkolab. On preemptive resource constrained scheduling: Polynomial-time approximation schemes. In *Integer Programming and Combinatorial Optimization*, volume 2337 of *Lecture Notes in Computer Science*, pages 329–349. Springer Berlin / Heidelberg, 2006.

[KAL96] J. Korst, E. Aarts, and J. K. Lenstra. Scheduling periodic tasks. *INFORMS Journal on Computing*, 8:428–435, 1996.

[KALW91] J. Korst, E. Aarts, J. K. Lenstra, and J. Wessels. Periodic multiprocessor scheduling. In E. H. L. Aaarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91 Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, pages 166–178. 1991.

[Kel08] H. Kellerer. An approximation algorithm for identical parallel machine scheduling with resource dependent processing times. *OR Letters*, 36:157–159, 2008.

[LGW08] K.-H. Loh, B. Golden, and E. Wasil. Solving the one-dimensional bin packing problem with a weight annealing heuristic. *Computers & Operations Research*, 35:2283–2291, 2008.

[LKA04] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[LST90] J. K. Lenstra, David B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. In *Mathematical Programming 46*, pages 259–271, 1990.

[LT99] T. W. Lam and K. K. To. Trade-offs between speed and processor in hard-deadline scheduling. In *Proceedings of the tenth annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 623–632, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[Mok76] A. Mok. Task scheduling in the control robotics environment. Technical Report TM-77, 1976.

[MT90] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, revised edition, November 1990.

[New80] D. J. Newman. Simple analytic proof of the prime number theorem. *American Mathematical Monthly*, 87:693–696, 1980.

[NW12] Martin Niemeier and Andreas Wiese. Scheduling with an Orthogonal Resource Constraint. In *10th Workshop on Approximation and Online Algorithms (WAOA2012)*, 2012.

[Pap94] Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[PSTW97] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 140–149, New York, NY, USA, 1997. ACM.

[Sah76] S. Sahni. Algorithms for scheduling independent tasks. *Journal of the ACM*, 23:116–127, 1976.

[SKJ97] A. Scholl, R. Klein, and C. Jürgens. BISON: a fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & Operations Research*, 24:627–645, 1997.

[SL94] D. Simchi-Levi. New worst-case results for the bin-packing problem. *Naval Research Logistics*, 41:579–585, 1994.

[ST93] D. B. Shmoys and É. Tardos. Scheduling unrelated machines with costs. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1993)*, pages 448–454. ACM, 1993.

[Van99] F. Vanderbeck. Computational study of a column generation algorithm for bin packing and cutting stock problems. *Mathematical Programming*, 86:565–594, 1999.

[Vaz01] V.V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[VBJN94] P. H. Vance, C. Barnhart, E. L. Johnson, and G. L. Nemhauser. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications*, 3:111–130, 1994.

[VdC99] J. M. Valério de Carvalho. Exact solution of bin packing problems using column generation and branch and bound. *Annals of Operations Research*, 86:629–659, 1999.

[WL83] W. D. Wei and C. L. Liu. On a periodic maintenance problem. *Operations Research Letters*, 2:90–93, 1983.

[Woe97] G. J. Woeginger. There is no asymptotic ptas for two-dimensional vector packing. *Information Processing Letters*, 64(6):293 – 297, 1997.

[Wol98] L.A. Wolsey. *Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, 1998.

[Zuc07] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3:103–128, 2007.

Martin Niemeier

Contact Information

Dortmunder Str. 14 | *Mobile:* +49-(0)176 50352882
10555 Berlin | *E-mail:* martin.niemeier@epfl.ch
Germany | *WWW:* www.math.tu-berlin.de/?niemeier

Personal data

Date of birth: 28th February 1983
Nationality: German
Other Languages: English (fluent), French (basic)

Education

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Ph.D. candidate, Mathematics, since November 2008

- Thesis Topic: *Approximation algorithms for modern multi-processor scheduling problems*
- Adviser: Professor Friedrich Eisenbrand
- Area of Study: Approximation algorithms/Scheduling problems/Polyhedral geometry

University of Paderborn, Paderborn, Germany

Diplom, Mathematics, September 2008
Diplom, Computer Science, September 2008

- Thesis Topic: *Integer decomposition for quasi-line graphs*
- Adviser: Professor Friedrich Eisenbrand

B.S., Computer Science, Juli 2007

Scientific Publications

[1] Martin Niemeier and Andreas Wiese. Scheduling with an Orthogonal Resource Constraint. In *10th Workshop on Approximation and Online Algorithms (WAOA2012)*, 2012.

[2] Nicolas Bonifas, Marco Di Summa, Friedrich Eisenbrand, Nicolai Hähnle, and Martin Niemeier. On sub-determinants and the diameter of polyhedra. In *28th Symposium on Computational Geometry (SoCG 2012)*, 2012.

[3] Friedrich Eisenbrand and Martin Niemeier. Coloring fuzzy circular interval graphs. *European Journal of Combinatorics*, 2011.

[4] Sanjoy Baruah, Martin Niemeier, and Andreas Wiese. Partitioned real-time scheduling on heterogeneous shared-memory multiprocessors. In *23rd Euromicro Conference on Real-Time Systems (ECRTS2011)*, 2011.

[5] Friedrich Eisenbrand, Nicolai Hähnle, and Martin Niemeier. Covering Cubes and the Closest Vector Problem. In *27th Symposium on Computational Geometry (SoCG 2011)*, 2011.

[6] Friedrich Eisenbrand, Nicolai Hähnle, Martin Niemeier, Martin Skutella, José Verschae, and Andreas Wiese. Scheduling periodic tasks in a hard real-time environment. In *37th International Colloquium on Automata, Languages and Programming (ICALP2010)*, 2010.

[7] Friedrich Eisenbrand, Karthikeyan Kesavan, Raju Mattikalli, Martin Niemeier, Arnold Nordsieck, Martin Skutella, José Verschae, and Andreas Wiese. Solving an Avionics Real-Time Scheduling Problem by Advanced IP-Methods. In *18th Annual European Symposium on Algorithms (ESA2010)*, 2010.

[8] Friedrich Eisenbrand and Martin Niemeier. Coloring Fuzzy Circular Interval Graphs. In *European Conference on Combinatorics, Graph Theory and Applications (EuroComb2009)*, 2009.

Teaching Experience

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Teaching Assistant — **February 2009 to May 2011**

- Spring semester 2011
  - Optimisation Discrète (assistant)
  - Algèbre linéaire 2 (assistant)
- Fall semester 2010
  - Algèbre linéaire 1 (assistant)
- Spring semester 2010
  - Discrete Optimization (main assistant)
- Fall semester 2009
  - Combinatorial Optimization (main assistant)
- Spring semester 2009
  - Introduction To Discrete Optimization (main assistant)

University of Paderborn, Paderborn, Germany

Student Assistant — **Fall 2007**

- Mathematik für Informatiker 1 (Analysis)

Professional Experience

Technische Universität Berlin, Berlin, Germany

Wissenschaftlicher Mitarbeiter — **March 2012 to present**

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Doctoral Assistant — **November 2008 to February 2012**

University of Paderborn, Paderborn, Germany

Student assistant (SHK) — **October 2007 to February 2008**

- Teaching assistant

Novatec Kommunikationstechnik, Paderborn, Germany

Student assistant — **March 2006 to March 2007**

- Software development for embedded telecommunication systems