APPROXIMATE FEASIBILITY IN REAL-TIME SCHEDULING

Speeding up in order to meet deadlines

ISBN 978 90 361 0395 4

Cover design: Crasborn Graphic Designers bno, Valkenburg a.d. Geul
and Ay May Ho (image)

This book is no. **586** of the Tinbergen Institute Research Series, established through cooperation between Thela Thesis and the Tinbergen Institute. A list of books which already appeared in the series can be found in the back.

VRIJE UNIVERSITEIT

Approximate feasibility in real-time scheduling

Speeding up in order to meet deadlines

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. F.A. van der Duyn Schouten,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Economische Wetenschappen en Bedrijfskunde
op maandag 23 juni 2014 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door
Suzanne Laura van der Ster
geboren te Amsterdam

promotoren: prof.dr. L. Stougie
prof. A. Marchetti-Spaccamela

Contents

Chapter 1

Introduction

Imagine you are a PhD candidate in the Netherlands and your thesis defence is approaching. Of course, you asked two friends to be your paranymphs.[1] Apart from accompanying you at your thesis defence, they agreed to help you with preparations for the party that you are having that night. Before your defence, your paranymphs will make sure the party location is ready. You gave your paranymphs the following list of jobs, each with the time that this job should take.

putting up bar tables	25 min
beer engine ready	15 min
polishing glasses	20 min
decorating the room	20 min
picking up snacks from shop	20 min

The paranymphs decided they will divide these jobs such that each job is performed by only one of them. They can leave the party location no earlier than that all jobs are done, so they will have to figure out a good way to divide the jobs. Note that there is no feasible way to divide the jobs such that both paranymphs are done at the same time. So, for the given settting, the best division is that one of them performs the job that takes 25 minutes (putting up bar tables) and one of the jobs taking 20 minutes (polishing glasses, for example). The other paranymph then performs the other two jobs taking 20 minutes (decorating the room and picking up the snacks from the shop), and the job taking 15 minutes (preparing the beer engine). The paranymph to finish last will be finished after 55 minutes.

The problem given above is an example of a *scheduling problem*. In scheduling, we call the moment of finishing the last job the *makespan* of a schedule. For the setting where each of the jobs can only be assigned to one of the paranymphs, the schedule as given above is *optimal*, i.e., no schedule exists, giving a smaller makespan. Problems arising in

[1] A *paranymph* is a ceremonial assistant, originally (in ancient Greece) for the bride and groom at a wedding. In Dutch doctoral thesis defences, it is customary that the candidate has two paranymphs accompanying him or her. In former times, the paranymphs had to be able to physically defend the candidate but the doctoral candidate could also ask them for advice when answering questions. Although officially paranymphs are still allowed to answer questions for the candidate, today their role is mainly symbolic.

the field of scheduling all have in common that they are searching for the best way to commit a set of resources to a set of jobs to be performed. What is "the best way" may differ in each of the problems, and also the constraints that the sought-after schedule has to obey are different among different problems.

This dissertation consists of four different scheduling problems. Later in this chapter we will elaborate more on the problems that are considered in this dissertation and how they are related to one another. First, we will more formally define the theory of scheduling.

The field of scheduling consists of problems in the *classical scheduling* field and problems from *real-time scheduling*. We will introduce the classical scheduling setting first, as the real-time setting is a generalization of that.

1.1 Classical scheduling theory

A typical scheduling problem consists of a set of n jobs, each of them having a certain *processing requirement*. In order to process the jobs, we are given one or more resources, that we will call *machines*. Whereas the word "machine" may evoke thoughts of the classical machine in a factory, a machine in the scheduling model could also represent a processor in a computer or a person (a paranymph, in the above example) performing a set of jobs. Then, we need to specify what kind of schedule we are aiming for. So we need to specify an objective function that is minimized (or maximized) and, if necessary, define the other problem parameters.

Consider the following example instance of a scheduling problem.

Example 1. *Five jobs are given and each job j has a* processing requirement p_j. *The processing requirements of jobs* $1, \ldots, 5$ *are* 3, 4, 6, 7 *and* 10, *respectively. There are two machines to process these jobs and each machine can process one job at the time. There are various ways to schedule the jobs on the machines. Let us first assume that we want to minimize the time instant where the latest job finishes. This measure is called the* makespan. *It is not hard to see that we cannot find a schedule with makespan less than* 16. *In Figure 1.1, a schematic representation of one of the schedules yielding this makespan is given. Each row represents one machine and the width of each block represents the processing requirement of the corresponding job.*

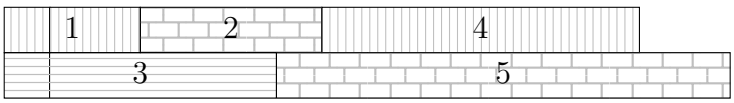

Figure 1.1: Gantt chart depicting one of the optimal schedules for makespan minimization.

Another possible objective to minimize is the sum of completion times. *That is, for all jobs we denote the amount of time that has elapsed since the start of the schedule, until the completion of the job. Those completion times are then summed. The Shortest Processing Time first (SPT) rule, that orders jobs in non-decreasing order of processing times and assigns in this order the jobs to a machine, whenever one becomes available, is optimal [54] for this problem. Figure 1.2 shows the optimal schedule resulting from*

applying this rule. Note that schedule is very different from the schedule in Figure 1.1 and is not balanced as nicely.

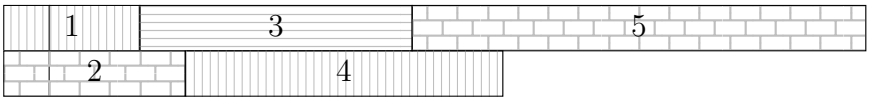

Figure 1.2: Gantt chart depicting the optimal schedule for minimizing the sum of completion times.

Three-field notation In classical scheduling theory, to efficiently denote all parameters of a scheduling problem, the three-field scheduling notation by Graham et al. [42] is used. A scheduling problem is denoted in the form $\alpha|\beta|\gamma$, where parameter α denotes the machine environment, the parameter β the job characteristics and γ denotes the objective function.

For the machine environment there are many possible settings, of which only those relevant for this dissertation will be explained here. If $\alpha = P$, we have *parallel* machines. This means that the machines are identical and the processing requirement of job j can be denoted by p_j for all machines. If $\alpha = Q$, we speak of *uniform* machines (or sometimes, related machines). Each machine i now has a speed σ_i and the processing requirement of job j on machine i is p_j/σ_i. Consider again the example given in the beginning of this chapter, and suppose your little brother wants to help the paranymphs preparing your party. He is only twelve years old, and not working as efficiently as the paranymphs. This can be modeled by a machine working at a speed that is half of the speed at which the paranymphs are working. Whereas in this setting the dependence of the processing requirement on the machine is uniform over all jobs, in the setting of *unrelated* machines ($\alpha = R$) this dependence is arbitrary. The processing requirement of job j on machine i is then denoted by p_{ij}. This setting can model situations where machines are specialized and thus better at certain jobs (and not so good at others). If the number of machines in the problem is specifically given, we write this number after the parameter specifying the machine environment (otherwise we assume an arbitrary number m of machines). Note that if we have just one machine, all of these environments boil down to the same and we will just write $\alpha = 1$.

In the field of job characteristics, we can encounter more than one parameter at the same time. We will list the parameters relevant for this dissertation. The term "pmtn" denotes *preemption*. If preemption is allowed, it means that after processing of a certain job has started, the job may be interrupted and resumed later (even on a different machine). It is, however, not allowed to process parts of the same job simultaneously on more machines. This is allowed in the setting of job splitting ("split"). Note that the paranymphs from our example chose for a non-preemptive schedule. They could have improved their schedule by allowing preemption or job splitting. We can add *setup times* to the problem which means that before each job (part) can be scheduled, the machine is unavailable for some amount of time, because it is setting up for the new job. The symbol "s" denotes the presence of *uniform* setup times, i.e., the setup time is independent of the machine that is setting up, or the job that it is setting up for. Further, each job j

may have a *release date* ("r_j"), meaning that the job is not available for processing prior to time r_j. In case the presence of release dates is not stated in field β, it is assumed that $r_j = 0$ for all jobs j. The symbol "d_j" expresses the presence of *due dates*. In real-time scheduling a due date is usually called a *deadline*.

The most common objective functions in classical scheduling are the makespan, the sum of completion times and the weighted sum of completion times. Let us denote by C_j the completion time of job j, i.e., the moment in time that job j completes in a given schedule. The makespan $C_{\max} := \max_j C_j$ is then defined as the maximum over all completion times, i.e., the moment that the last job completes. The sum of completion times is simply defined as $\sum_j C_j$. Supposing that each job j is given a *weight* w_j expressing its importance, the weighted sum of completion times is then defined as $\sum_j w_j C_j$.

1.2 Real-time scheduling

Real-time scheduling is a generalization of the classical scheduling theory. Whereas in classical scheduling we deal with a finite set of jobs, in real-time scheduling we typically deal with an infinite amount of jobs, that are released by so-called *sporadic tasks* in a structured manner. A task is a piece of code that generates multiple jobs over time. A task τ is characterized by an execution requirement c_τ, a (relative) deadline d_τ and a period p_τ, that denotes the minimum interarrival time between two consecutive jobs from the same task. If a task τ releases a job at time a, the job needs c_τ units of execution before its absolute deadline that is at $a + d_\tau$. The next arrival of a job from the same task can be no earlier than $a + p_\tau$. A *task system* $\mathcal{T}$ consists of n tasks $\tau_1, \ldots, \tau_n$. We call a job *active* if it has been released but not yet completed.

If a machine executing a sporadic task system is said to run at speed σ, we assume that it can execute σ units of work per time unit, instead of one. Note that this is equivalent to running the task system with all execution requirements scaled by a factor $1/\sigma$ on a unit-speed machine.

For a task τ we define the *utilization* of that task as $u_\tau = \frac{c_\tau}{p_\tau}$, the maximum execution requirement per time unit of jobs from this task.

We classify task systems according to the relation that exists between deadlines and periods of its tasks. If for all tasks the deadline equals the period ($d_\tau = p_\tau$, for all τ), we call the system an *implicit-deadline* task system. If this is not the case but for all τ it holds that $d_\tau \leq p_\tau$, we have a *constrained-deadline* system. If $d_\tau > p_\tau$ for at least one of the tasks τ, we say the system has *arbitrary deadlines*.

1.2.1 Feasibility

In real-time scheduling, the focus is not on finding a schedule that optimizes a certain objective function, but on determining whether a schedule can be found such that all jobs from all tasks meet their deadlines.

Definition 1. *A task system is said to be* feasible *on a computing platform if, for any possible job sequence generated by the system, there exists a schedule for the system, such that all jobs from all tasks meet their deadlines.*

For scheduling a task system on a single machine, much is known about determining feasibility. The tightest case is when the first jobs of all tasks arrive simultaneously (at time 0) and all subsequent jobs arrive as soon as permitted by the period parameters (i.e., task τ generates a job at each time instant $k\,p_\tau$, for $k = 0, 1, 2, \ldots$) [21]. We call this sequence of job arrivals the *synchronous arrival sequence*. Further, if a task system is feasible, the Earliest Deadline First (EDF) scheduling policy, that schedules at any time the active job with the earliest absolute deadline, will produce a valid schedule [56].

Determining feasibility of an implicit-deadline task system is easy. It is well known that an implicit-deadline task system is feasible on a speed-σ processor if and only if the sum over the utilizations of all tasks is at most σ. The following proposition formalizes this statement.

Proposition 1 (Liu and Layland [56])**.** *For an implicit-deadline task system $\mathcal{T}$, the Earliest Deadline First algorithm is a correct scheduling policy for a processor of speed σ if and only if $\sum_{\tau\in\mathcal{T}} u_\tau \leq \sigma$.*

For arbitrary-deadline systems, determining feasibility is more complicated. It is known [21] that a set of sporadic tasks $\mathcal{T}$ is EDF-schedulable on a unit-speed machine if and only if the following conditions are satisfied:

1. the utilization of the task system does not exceed 1, i.e., $\sum_{\tau\in\mathcal{T}} u_\tau \leq 1$,
2. all jobs with deadlines $[0, lcm_{\tau\in\mathcal{T}}(p_\tau)]$ in the synchronous arrival sequence of $\mathcal{T}$ meet their deadlines (where lcm denotes the least common multiple).

This immediately yields an exponential-time test to check whether $\mathcal{T}$ is EDF-schedulable.

The two conditions given above can be translated into one function, called the *demand bound function*. We define the function

$$dbf_\tau(t) = \max\left\{0, \left\lfloor \frac{t + p_\tau - d_\tau}{p_\tau} \right\rfloor c_\tau\right\} \tag{1.1}$$

as the demand bound function of task τ at time t. It represents the (maximum) total workload of the jobs generated by τ that needs to be finished by time t. For a task system $\mathcal{T}$ to be feasible, the total workload of the jobs generated by all tasks in $\mathcal{T}$ up to time t can not exceed the amount of work the processor can perform up to time t. This gives us the following necessary and sufficient condition for feasibility of a task system $\mathcal{T}$.

Proposition 2 (Baruah, Mok and Rosier [21])**.** *A task system $\mathcal{T}$ is feasible on a preemptive processor running at unit speed if and only if*

$$dbf_\mathcal{T}(t) := \sum_{\tau\in\mathcal{T}:d_\tau\leq t} \left\lfloor \frac{t + p_\tau - d_\tau}{p_\tau} \right\rfloor c_\tau \leq t, \qquad \forall t \geq 0. \tag{1.2}$$

1.3 Complexity and approximation

1.3.1 Complexity classes

Complexity theory is a central field of the theoretical foundations of computer science and focuses on classifying computational problems according to their intrinsic complexity. Rather than giving an "absolute" answer about the complexity of computational

problems, complexity theory has been more successful at classifying problems to their relative complexity.

Decision problems A *decision problem* is a computational problem that can only be answered with 'yes' or 'no'. For example, given a number of machines and some jobs with processing requirements to schedule on them; does there exist a non-preemptive schedule for these jobs with makespan at most C? This problem is the decision version of the MAKESPAN MINIMIZATION problem, that we have seen earlier in this chapter. Any other combinatorial optimization problem can also be formulated as a decision problem. These decision problems form the basis of complexity theory.

Problem instance A computational problem like the makespan minimization problem can be viewed as an infinite collection of *problem instances*. The example given in the beginning of this chapter, consisting of two machines (the paranymphs) and a list of jobs with processing times is one particular input to the problem, and is called an instance of the makespan minimization problem.

We denote by $|I|$ the *size* of an instance I. The size of an instance is the number of binary information bits needed to represent the instance. In scheduling problem instances we usually assume that the number of jobs (or tasks), n, and the number of machines, m, should be part of the input size. However, it might be the case that the input parameters (e.g., processing times) are so large, that they do not fit into the space of a "computer word" (which has a fixed size)[2] and they should be part of the input size as well. Representing each number as a binary number (base 2), and if the processing times can be very large, then for an instance of the unrelated-machines problem (see Chapter 3) the input size would be $O(nm \log p_{\max})$, where $p_{\max} = \max_{i,j} p_{ij}$.

Algorithm An algorithm is a step-by-step procedure to solve a computational problem. It outputs the optimal solution for a problem instance, or outputs that it cannot find any solution. (Note that there are also algorithms that do not output the optimal solution, but a solution that is sufficiently close to the optimal solution. These algorithms are called approximation algorithms and they are more formally defined in Section 1.3.2.)

Definition 2. *An algorithm is said to run in* polynomial time *if for any instance I the number of computational steps needed to solve I is bounded by a polynomial in the size $|I|$ of the instance.*

The class NP Suppose, given an instance of the decision problem described above, the answer is 'yes'. Given a schedule, it is very easy to verify that it is a correct schedule (all jobs are scheduled) and it gives the appropriate makespan. Actually, the maximum number of steps required to check this, is polynomial in the size of the problem instance. This brings us to a very natural class of problems: the class NP.

[2]If each input parameter fits into a computer word, the number of symbols necessary to represent each of those number is bounded by a constant, and we can ignore the numbers when describing the input size. If the number do not fit into this fixed size, then we need to take them into account explicitly.

The class NP is the collection of all decision problems for which each instance with a *positive* answer has a polynomial-time checkable *certificate* of correctness. In the case of makespan minimization, the certificate consists of a partitioning of the jobs over the machines. The check consists of adding the processing times of jobs assigned to each machine (and check that the sum is at most C) and checking that all jobs are assigned to exactly one machine.

Note that for the complementary question (Given a set of machines and a set of jobs with processing requirements; does no schedule exist with makespan at most C?), no polynomial-time checkable certificate for a positive answer is known to exist. Any certificate giving a schedule with greater makespan does not prove that no schedule with makespan at most C exists. This decision problem is in the class co-NP. This class exists of all problems, for which the complementary problem belongs to NP.

NP-complete problems Although a certificate for the decision version of the makespan minimization problem can be checked in polynomial time, it is not known how to find a schedule with makespan at most C in polynomial time [24]. The decision version of the makespan minimization problem belongs to the hardest problems within the class NP. Problems that are at least as hard as any problem in NP, are called NP-hard problems (note that an NP-hard problem is not necessarily in NP, though). Problems that are both in NP and are NP-hard (e.g., the makespan minimization problem), are called NP-complete.

The class P As mentioned above, it is not known, for an arbitrary scheduling instance, how to find a schedule of makespan at most C in polynomial time. However, for some problems that are in NP, it is possible to construct the certificate for a positive answer in polynomial time. For example, for the problem of minimizing total completion time (a different term for the sum of completion times).

The decision version of the total completion time problem is as follows: given a set of machines and a set of jobs with processing requirements, does there exist a (non-preemptive) schedule such that the total completion time is at most $\bar{C}$? As mentioned earlier in this chapter, the Shortest Processing Time first rule gives the minimum total completion time [54]. This algorithm can thus be used to answer this question for any instance of the problem, in polynomial time.

The class P is a subset of NP and consists of all problems for which a polynomial-time algorithm exists solving any instance of this problem.

Reductions To prove that a problem belongs to P, it is sufficient to find a polynomial-time algorithm solving the problem. Intuitively, it is not so easy how to show that a problem is NP-complete. Clearly, it is not sufficient to claim that everything was tried to find a polynomial-time algorithm and this failed, and thus assume that probably no polynomial-time algorithm exists.

To place problems in a certain complexity class, we use the technique of *reduction*. This is a transformation from one problem into another under certain conditions. If there exists a mapping from any instance I of problem Π to an instance I' of problem Π' that

can be computed in polynomial time, and I is a 'yes' instance of Π if and only if I' is a 'yes' instance of Π', we say that Π polynomially reduces to Π'. This implies that problem Π is not harder than Π'. Polynomial-time reductions are the core tool used for classifying problems.

To prove that problem X is NP-complete, we take a problem that is known to be NP-complete (say problem Y) and reduce it to X in polynomial time. Then, Y is no harder than problem X, but since Y is NP-complete, problem X must be at least NP-complete as well. More general, problem X is NP-complete if *all* problems in NP can be polynomially reduced to X and X is in NP.

The open question in complexity theory is the "P versus NP" question. Although we made a distinction between problems that are in P and that are NP-complete, it is not proven that those sets are distinct. If someone would ever prove one of the NP-complete problems to be easy to solve after all (i.e, to belong to P), then so are all problems in NP. This scenario, however, is deemed highly unlikely by almost any mathematician and computer scientist.

Any statements in this dissertation about hardness of the problems considered, are made under the assumption that $P \neq NP$.

1.3.2 Approximation

Many of the problems we consider in this dissertation are NP-hard, and therefore it is unlikely to find polynomial-time algorithms to solve them. When studying such problems, the best we can do in polynomial time, is to find algorithms that give a solution that is "close enough" to the optimal solution. Algorithms that turn out to work well on average (or in practice), but that have no proven bound on how far away their solutions are from the optimal solution, we usually call *heuristics*. If such an algorithm returns a solution that is guaranteed to be within some multiplicative factor α from the optimal solution, we call it an *approximation algorithm*. The factor α is called the *approximation ratio*. The closer the approximation ratio is to 1, the better the approximation algorithm is.

An approximation algorithm is called a PTAS (polynomial-time approximation scheme) if its approximation ratio equals $1+\epsilon$, for any small $\epsilon > 0$, and it runs in polynomial time. The running time is also dependent on $1/\epsilon$, but this dependence may be exponential. If the running-time dependence on $1/\epsilon$ is only polynomial, we call the algorithm a fully polynomial-time approximation scheme (FPTAS).

In Chapters 3, 4 and 5, where we are interested in checking feasibility of task systems, the notion of *approximate feasibility tests* will appear. Since in the context of a feasibility test there is no optimization, the approximation will be in the sense of the *resource augmentation setting*. Although it was much earlier used in the context of online algorithms for the paging problem (see, e.g., [67]), this notion was first explicitly introduced for scheduling problems in [46] as a method for comparing worst-case behavior of different algorithms for solving the same problem. If an α-approximate feasibility test returns "feasible", the task system is guaranteed to be feasible on a processor that runs at speed α, while if it returns "infeasible", the task set is guaranteed to be infeasible when processed on a unit-speed processor. The factor α is also called the *speedup factor*. When designing approximate feasibility tests, the aim is to obtain a speedup factor as close to

1 as possible.

Analogous to the setting of approximation algorithms, the notions of PTAS and FPTAS also exist when dealing with feasibility tests. A PTAS feasibility test has a speedup factor of $1+\epsilon$ and runs in polynomial time in the instance size, but the running-time dependence on $1/\epsilon$ is super-polynomial. If this running-time dependence on $1/\epsilon$ is polynomial, we have an FPTAS feasibility test.

1.4 Outline

In this dissertation, both classical scheduling and real-time scheduling problems are present. The outline of the dissertation is as follows.

Chapter 2 is the only chapter dealing with a scheduling problem in the classical setting. We consider a number of parallel identical machines and a job set to be scheduled upon these machines. The objective is to minimize the sum of completion times. Jobs can be split into parts and parts of the same job can be processed simultaneously. However, before each job (part) can be processed, a uniform setup time s is required on each machine. The main result in this chapter is a polynomial-time algorithm solving the problem for two machines. For more machines, the problem is more complicated and we do not know the complexity of this problem. We do give a very simple $2 + \frac{1}{4}(\sqrt{17} - 1) \approx 2.781$-approximation algorithm for the general case.

The other three chapters consider problems from the real-time paradigm. Chapters 3 and 4 are quite closely related. Both chapters consider a set of real-time tasks and multiple machines. The goal is to find a partitioning of the task set over the machines such that all jobs released by a specific task are executed on the machine that this task is assigned to, such that all deadlines are met.

In Chapter 3 the machines are unrelated. We give a $8 + 2\sqrt{6} \approx 12.9$ approximation for the partitioning problem on an arbitrary number of machines. In order to obtain this result we introduce linear approximations of the demand bound function and on top of that, we develop a rounding procedure that is of independent interest and is given in a separate section. We also give a polynomial-time approximation scheme for the case that the number of machines m is a constant. For a fixed $\epsilon > 0$, the test decides in time polynomial in the number of tasks whether there exists a partition of the task set over the machines such that the task set can be feasibly scheduled if the machines run at speed $1 + O(\epsilon)$ or whether no feasible partition exists if the machines run at unit speed.

In Chapter 4 we consider parallel identical machines. Building upon ideas from the PTAS in Chapter 3 we find a $(1 + \epsilon)$-approximate feasibility test for the problem of partitioning a task set over an arbitrary number of identical machines. We improve upon a result by Chen and Chakraborty [30] who give a PTAS for the case that the ratio of the maximum relative deadline to the minimum relative deadline $d_{\max}/d_{\min}$ is bounded by a constant. Denoting this ratio by λ, the running time of their algorithm is roughly $n^{O(\exp(\frac{1}{\epsilon}\log\lambda))}$, hence doubly exponential in λ. Our approximation scheme runs in $O(m^{O(f(\epsilon)\log\lambda)})$ time, where $f(\epsilon)$ is a function depending solely on ϵ, and hence an exponential improvement over the result from [30] is given.

Finally, Chapter 5 also deals with real-time tasks but considers the *mixed-criticality*

setting. In this setting, the task system is always in one out of K levels (think of the levels as states that the system can be in), and each task has different execution-time parameters for different levels. We will delay a detailed description of this setting to Chapter 5, because it requires many concepts and a lengthy explanation that are not relevant to any other chapter in this dissertation. We consider feasibly scheduling an implicit-deadline mixed-criticality task system on a single machine. We give a scheduling algorithm EDF-VD that decides in polynomial time that either a mixed-criticality task system consisting of K levels ($K \in \mathbb{N}$) can be scheduled upon a processor running at a faster speed or that it cannot be scheduled on a processor running at unit speed. The required speedup depends on the number of levels K. For any 2-level implicit-deadline task system we show that this speedup is at most $4/3$ and for 3 levels we show a speedup factor of 2. Finally we show that no (non-clairvoyant[3]) algorithm can guarantee correctness on a processor with speedup less than $4/3$. For dual-criticality systems we give a fixed-priority policy RM-VP needing a speedup of $\phi / \ln 2 \approx 2.334$ (where ϕ equals the golden ratio). Finally, for scheduling a mixed-criticality job set on m parallel machines we give a fixed-priority scheduling policy called OCBP, needing a speedup of $\phi + 1 - \frac{1}{m}$.

[3]Meaning that it is not known in advance to the algorithm in which level the system will be. See Section 5.2 for a formal definition of clairvoyance.

Chapter 2

Split scheduling

Almost all results in this chapter are based on work that appeared in [63].

2.1 Introduction

In this chapter we consider a classical scheduling problem, in a setting with *job splitting* and *setup times*. We consider multiple parallel machines and our objective is to minimize the sum of completion times. Whereas in the setting of ordinary preemption it is not allowed that multiple machines process the same job simultaneously, in job splitting this constraint is dropped. So each job can be split into multiple parts and machines may process these parts simultaneously. However, before the processing of a job (part) can start, a setup time is required.

2.1.1 Related work

Without setup times, many scheduling problems become trivial if preemption is replaced by job splitting. For example, for minimizing the sum of completion times, it would be optimal to split all jobs equally over all machines and process them in SPT order. For minimizing the makespan, the order would not even matter. In the presence of release times, minimizing the sum of completion times with ordinary preemption is NP-hard [33], but if job splitting is allowed, it is not hard to see that splitting all jobs over all machines and processing them in Shortest Remaining Processing Time first (SRPT) order is optimal. We refer to Xing and Zhang [73] for an overview of several classical scheduling problems which become polynomially solvable if job splitting is allowed.

If setup times enter the stage, triviality disappears. Then, before starting processing of a job (part), the machine requires a setup time, during which it cannot process another job (part), nor set up processing for any other job (part). Problems in which setup times are assumed to be sequence-dependent, are usually NP-hard, as such problems exhibit routing-like features. For example, the Hamiltonian path problem in a graph can be reduced to the problem of minimizing the makespan on a single machine, where each job corresponds to a node in the graph, all processing times are 1 and the setup time between job j and k is 0 if the graph contains an edge between nodes j and k, and 1 otherwise.

We encountered problems in this setting by studying disaster relief operations [70], such as flood relief operations, where machines are pumps and jobs are locations to be drained. In the case of earthquake relief operations the machines represent teams of relief workers and the jobs locations to be cleared. The setup time then represents the time that is needed to instruct a new team about a location or move water pumps (overnight) to a new location. Although in practice these setup times consist partly of travel time, the main part of the setup consists of equipping teams with tools and instructions for a new location. Hence the decision was made that considering job-, machine- and sequence-independent setup times was an acceptable approximation of reality. As we will see in this chapter, making this assumption about the setup times, still presents us with algorithmically challenging problems.

From now on, we will refer to setup times that are job-, machine- and sequence-independent as *uniform setup times*. We denote the setup time by the parameter s. Not much literature exists on scheduling problems with such setup times. The problem of minimizing the makespan on parallel identical machines, with job splitting and setup times that are job-dependent, but machine- and sequence-independent is studied by Xing and Zhang [73] and by Chen, Ye and Zhang [28]. Chen et al. [28] note that this problem is NP-hard in the strong sense, and only weakly NP-hard for the case of a constant number of machines. Straightforward reductions from the 3-Partition problem and Subset Sum problem show that these hardness results continue to hold if setup times are uniform. Chen et al. provide a 5/3-approximation algorithm for this problem and an FPTAS for the case of a constant number of machines. For the same problem but with preemption instead of job splitting, Schuurman and Woeginger [64] give a PTAS. It remains open whether a PTAS exists if not preemption, but job splitting is considered, even for uniform setup times. See [57] and [60] for more problems with preemption and setup times.

Recently, Correa et al. [31] studied the problem of minimizing the makespan on unrelated machines with job splitting and unrelated setup times (i.e., the setup times are machine- and job-dependent). They show a lower bound on the approximability of this problem of $\frac{e}{e-1} \approx 1.582$ and give a $1+\phi$ approximation (where $\phi = \frac{1+\sqrt{5}}{2}$ equals the golden ratio) based on the approach by Lenstra, Shmoys and Tardos [53] for minimizing the makespan on unrelated machines (see also Section 3.1 for this problem). For the restricted-assignment case (for all j, $p_{ij} \in \{p_j, \infty\}$ and $s_{ij} \in \{s_j, \infty\}$), they even give a 2 approximation.

The problem we consider is related to scheduling problems with malleable tasks. A malleable task may be scheduled on multiple machines simultaneously, and a function $f_j(k)$ is given that denotes the speed if task j is processed on k machines. If task j is processed on k machines for L units of time, then $f_j(k)L$ units of task j are completed. What we call job splitting can be translated into malleable tasks with linear speedups, i.e., the processing time required on k machines is $1/k$ times the processing time required on one machine. However, split scheduling with setup times can not be translated into the setting with malleable tasks, because of the discontinuity caused by the setup times. See Drozdowski [32] for an extensive overview of the literature on scheduling malleable tasks.

2.1.2 Our results

As mentioned above, this chapter focuses on the problem of minimizing the sum of completion times on identical machines, with job splitting and uniform setup times. The version of this problem with ordinary preemption is solved by the SPT rule; the option of preemption is not used by the optimum [54]. However, for job splitting the situation is much less straightforward. If s is extremely large, an optimal schedule would minimize the contribution of the setup times to the objective, and a job will only be split over more machines, if no other job is scheduled after it on these machines. If s, on the other hand, is very small (say 0), then each job is split over all machines and the jobs are scheduled in SPT order. For other values of s, however, it appears to be a non-trivial problem to decide how to schedule the jobs. Splitting a job over multiple machines decreases the completion time of this job, but due to the extra setup times, the total load on the machines increases and hence the completion times of later jobs are increased. We give an example instance to illustrate this.

Example 2. *Given are 3 machines and 6 jobs, with processing times* 1*,* 2*,* 3*,* 5*,* 11 *and* 12*, respectively. Setting up a machine takes 1 time unit. One could start with filling up the machines in SPT order without any splits. This schedule is given in the Gantt chart in Figure 2.1(a). The objective value equals 49.*

This schedule can be improved by splitting job 6 over machines 1 and 3. This decreases

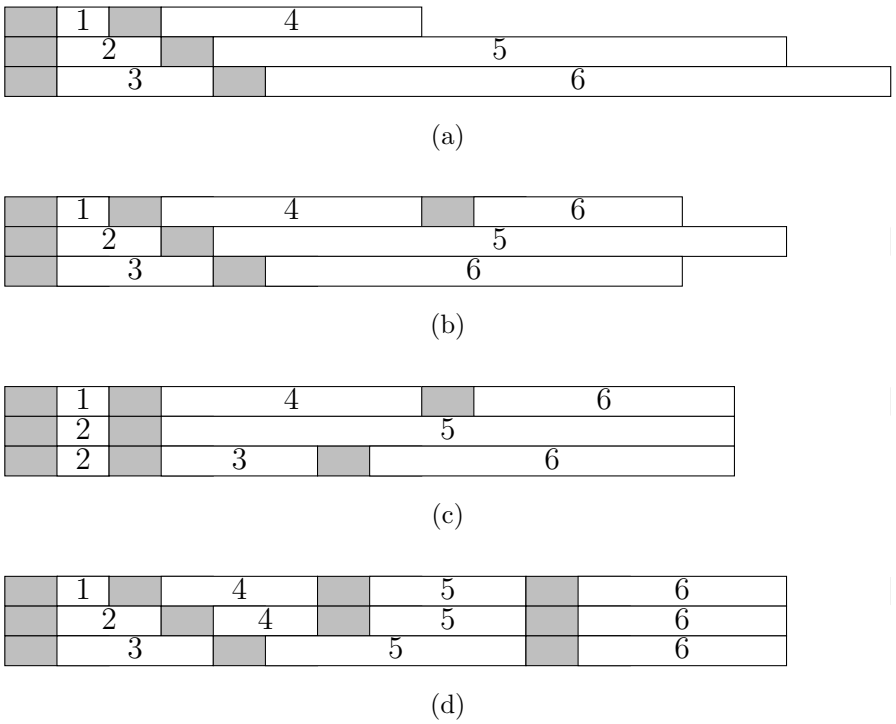

Figure 2.1: Gantt charts depicting the schedules for the example instance. The grey blocks indicate the setup times, the numbered blocks are scheduled job parts. Each row of blocks gives the schedule for one machine.

the completion time of job 6, and since no jobs are scheduled after job 6 on these machines, no other completion times increase. To make the biggest improvement in the objective value, both parts of job 6 should finish simultaneously. See Figure 2.1(b) for the Gantt chart of this improved schedule, with objective value 45.

Splitting jobs that appear earlier in the schedule may actually increase the objective value, as many later jobs experience delays due to extra setup times incurred. For example, if we choose to split job 2 over machines 2 and 3, we cause delays for jobs 3 and 6, while improving the completion times of jobs 2 and 5. If we require that all job parts of the same job finish at the same time, we obtain the schedule in Figure 2.1(c) with objective value 46. Finally, Figure 2.1(d) shows the optimal schedule, with objective value 40.

This example illustrates the inherent trade-off that makes this problem non-trivial. Splitting jobs will decrease the completion times of some jobs, but it may also increase the completion times of others, due to higher loads on the machines.

In Section 2.4 we give a polynomial-time algorithm to solve the special case of two machines. The algorithm is based on a careful analysis of the structure of an optimal solution. To this end, in Section 2.3 we give some properties that hold for the optimal solution on any number of machines. Then, in Section 2.4 we give additional properties that we prove for the case of two machines only. Together, these properties give us a handle to find in $O(n \log n)$ time the optimal solution for the two-machine case, where n is the number of jobs. Unfortunately, the additional properties given in Section 2.4 are not easily extended to more machines. In Section 2.5 we show that some of the nice properties that hold for the two-machine case already fail to hold for three machines. This fact bodes ill for finding an extension of the approach that we found for two machines. However, showing NP-hardness for this problem has also failed thus far. So it could be that we encountered another instance of Lawler's "mystical power of twoness" [52], a phrase signifying the surprisingly common occurrence that problems are easy when a problem parameter (here, the number of machines) is 2, but NP-hard when it is 3. But the possibility also exists that this problem is also easy for an arbitrary number of machines, and we just have not been able to see how to approach those cases. Note that if weights are added to the problem, i.e., the objective becomes minimizing the weighted sum of completion times, the problem is strongly NP-hard for three machines or more, and weakly NP-hard for the case of two machines [63].

In Section 2.6 we give a constant-factor approximation algorithm for the problem with an arbitrary number of machines, even though we are not sure the problem is NP-hard. The approximation algorithm is surprisingly simple; for each job the algorithm decides on how many machines it is scheduled, based only on the processing time of the job and the setup time s.

Although our main interest in this chapter lies in problems with uniform setup times, in Section 2.7 we study briefly two settings with non-uniform setup times. Both for machine-dependent and for job-dependent setup times, we discuss how much of the lemmas from Sections 2.3 and 2.4 remains valid.

The chapter is completed by an overview showing the state of the art for split scheduling problems with uniform setup times and showing which problems remain open.

2.2 Preliminaries

An instance is given by m parallel identical machines and n jobs. Job j has processing time p_j, for $j = 1, \ldots, n$. Each job may be split into parts and multiple parts of the same job may be processed simultaneously. Before a machine can start processing (a part of) a job, a uniform setup time s is required. During setup of a job (part), the machine cannot simultaneously process or setup another job (part). The objective we consider is to minimize the sum of completion times of the jobs (also called total completion time).

Given a schedule for the jobs, we denote by M_j the set of machines on which parts of job j are processed. We might sometimes say that a machine processes a certain job, when it only processes a part of that job. We call job j a d-job if $|M_j| = d$.

Given a schedule, we call a job *balanced* if it completes at the same time on all machines on which it is processed. By this definition, a 1-job is always balanced, so this characterization only has meaning for jobs j such that $|M_j| > 1$.

2.3 Properties of an optimal schedule

In this section we derive some properties of an optimal schedule, which are valid for any number of machines.

Claim 1. *Let $\mathcal{S}$ be a feasible schedule with job completion times $C_1 \leq C_2 \leq \ldots \leq C_n$. Let $\mathcal{S}'$ be obtained from $\mathcal{S}$ by rescheduling per machine the job parts in the order $1, 2, \ldots, n$, and let C' be the completion times in $\mathcal{S}'$. Then $C'_j \leq C_j$ for $j = 1, \ldots, n$.*

Proof. Let x_{ij} be the amount of time that job j is processed on machine i in $\mathcal{S}$ and let C_{ij} be the time that job j finishes on machine i. Let $y_{ij} = s + x_{ij}$ if $x_{ij} > 0$ and let $y_{ij} = 0$ otherwise. Fix some job j and machine i. Let $k = \arg\max\{C_{ik} \mid 1 \leq k \leq j\}$. Then $C_j \geq C_k \geq C_{ik} \geq \sum_{h=1}^{j} y_{ih} = C'_{ij}$, where the first inequality is by assumption and the last one by the definition of k (at time C_{ik} all work on jobs smaller than or equal to j has been done on machine i). Since $C_j \geq C'_{ij}$ for any machine i on which j is scheduled, the proof follows. □

This claim has several nice corollaries. First, note that if in an optimal schedule $C_1 \leq C_2 \leq \ldots \leq C_n$, then we maintain an optimal schedule with the same completion time for each job by scheduling the job parts on each machine in the order $1, 2, \ldots, n$. This allows to characterize an optimal schedule by a permutation of the jobs and the amount of time that job j is processed on each machine i, for all i and j. The optimal schedule is then obtained by adding a setup time s for each non-zero job part and processing the job parts in the order of the permutation on each machine. Consequently, in the optimal schedule obtained, each machine contains at most one part of each job.

We thus have the following lemma, which we will use throughout this chapter.

Lemma 1. *There exists an optimal schedule such that each machine contains at most one part of each job.*

Lemma 2. *There exists an optimal schedule that satisfies the property of Lemma 1 such that on each machine the job parts are processed (started and completed) in SPT order of the corresponding jobs.*

Proof. Among the optimal schedules that satisfy Lemma 1, we choose the schedule that minimizes $\sum_j p_j C_j$. By Claim 1, we may assume the jobs are numbered $1, \ldots, n$ so that $C_1 \leq C_2 \leq \ldots \leq C_n$, and each machine processes the job parts in the order given by the numbering of the jobs. By contradiction, suppose that there exist jobs k and ℓ such that $p_\ell < p_k$ and there is a machine i that processes job k before job ℓ, i.e., $C_k \leq C_\ell$. Choose among such pairs of jobs k and ℓ a pair that minimizes $\ell - k$. Note that any machine that processes both k and ℓ must process k immediately before ℓ, since if there is some job h that is processed between them, then $C_k \leq C_h \leq C_\ell$, and either $p_h > p_\ell$ or $p_h \leq p_\ell < p_k$, so either the pair ℓ and h or the pair k and h should have been chosen instead of the pair ℓ and k. We now show how to define a new optimal schedule for which $\sum_j p_j C_j$ is strictly less than for the original schedule, thus contradicting the choice of our schedule.

Note that $M_\ell \cap M_k \neq \emptyset$. We define a new schedule by rescheduling both jobs within the time slots these jobs occupy in the current schedule (including the slots for the setup times). First remove both jobs. Then consider the machines in M_k one by one, starting with the machines in $M_k \backslash M_\ell$ and fill up the slots previously used by job k with job ℓ, until we have completely scheduled job ℓ including the setup times. This is possible since $p_\ell < p_k$. We consider the remaining slots, which are single uninterrupted time intervals for each machine, by our choice of k and ℓ. We will show that they provide sufficient time for the processing and setup of job k, by showing that the combined number of setups for k and ℓ does not increase.

Let M'_k and M'_ℓ denote the sets of machines processing jobs k and ℓ, respectively, in the new schedule. We distinguish two cases. If job ℓ cannot be rescheduled completely in the slots used by k in $M_k \setminus M_\ell$ then we have $M'_k \subseteq M_\ell$. Together with $M'_\ell \subseteq M_k$ it follows that $(M'_k \cap M'_\ell) \subseteq (M_k \cap M_\ell)$. Hence, any machine containing both k and ℓ in the new schedule did also contain both jobs in the old schedule, and therefore there are no extra setups on any machine needed.

Now consider the case that job ℓ is rescheduled completely in the slots used by k in $M_k \setminus M_\ell$. Then, after adding job k, the total number of setups needed for ℓ and k does not increase since there is at most one machine of $M_k \setminus M_\ell$ containing both jobs in the new schedule, but none of the machines in $M_k \cap M_\ell$ is used by ℓ in the new schedule.

We conclude that the remaining slots after scheduling job ℓ provide sufficient room to feasibly schedule both the processing of job k and the required setups. Note that, if there is some machine on which the time allotted to job k is no more than s, then we can simply leave the machine idle for that time interval.

Let C' denote the new completion times. Since in the new schedule ℓ is processed only where job k was processed in the old schedule, and job k is processed in the new schedule only where either job ℓ or job k was processed in the old schedule, we have $C'_\ell \leq C_k$ and $C'_k \leq \max\{C_\ell, C_k\}$. By assumption, we have that $C_\ell \geq C_k$, and hence $C'_k \leq C_\ell$. Since for all other jobs the completion times remain the same, we have that the sum of completion times did not increase and since

$$\begin{aligned} p_k C_k + p_\ell C_\ell &= p_k(C_k - C'_k) + p_k C'_k + p_\ell(C_\ell - C'_\ell) + p_\ell C'_\ell \\ &> p_k C'_k + p_\ell C'_\ell + p_\ell(C_k - C'_\ell + C_\ell - C'_k) \\ &\geq p_k C'_k + p_\ell C'_\ell, \end{aligned}$$

we have that $\sum_j p_j C'_j < \sum_j p_j C_j$, which contradicts the choice of the original schedule. □

From now on, we assume that jobs are numbered in SPT order, i.e., $p_1 \leq \ldots \leq p_n$.

Lemma 3. *There exists an optimal schedule that satisfies the properties of Lemma 1 and Lemma 2 in which all jobs are balanced.*

Proof. Consider an optimal schedule of the form of Lemma 1 and Lemma 2 with a minimum number of job parts. Let C_j be the completion time of j in this schedule and define M_j for this schedule as before. Consider the following linear program in which there is a variable x_{ij} for all pairs i, j with $i \in M_j$, indicating the amount of processing time of job j assigned to machine i:

$$\begin{aligned}
&\min \sum_j C_j && \\
&\text{s.t.} \sum_{i \in M_j} x_{ij} = p_j && \text{for } j = 1, \ldots, n; \\
&\sum_{k \leq j:\ M_k \ni i} (s + x_{ik}) \leq C_j && \text{for } j = 1, \ldots, n,\ \forall i \in M_j; \\
&x_{ij} \geq 0, C_j \geq 0 && \text{for } j = 1, \ldots, n,\ \forall i \in M_j.
\end{aligned}$$

Note that a schedule that satisfies Lemmas 1 and 2 gives a feasible solution to the LP, and on the other hand that any feasible solution to the LP gives a schedule with total completion time at most the objective value of the LP: if there exist some j and $i \in M_j$ such that $x_{ij} = 0$, then the LP objective value is at least the total completion time of the corresponding schedule, as there is no need to set up for job j on machine i if $x_{ij} = 0$. We know that a solution is a basic solution to this LP, only if the number of variables that are non-zero is at most the number of linearly independent tight constraints (not including the non-negativity constraints). By the minimality assumption on the optimal schedule, in any optimal solution to the LP all C_j and x_{ij} variables are non-zero, which gives a total of $n + \sum_j |M_j|$ variables. Since there are only $n + \sum_j |M_j|$ constraints, all constraints must be tight, which proves the lemma. □

2.4 A polynomial-time algorithm for two machines

In this section we give more properties that hold for optimal schedules. However, these properties are shown only for the special case of two machines. Together with the properties from Section 2.3, these properties will lead to a polynomial-time algorithm for finding the optimal schedule in the case of two machines.

Lemma 4. *Let $\mathcal{S}$ be an optimal schedule for a two-machine instance that satisfies the properties of Lemmas 1, 2 and 3. Let $k < \ell$ be two consecutive 2-jobs. If there are 1-jobs between k and ℓ, then there is at least one 1-job on each machine. Also, the last 2-job is either not followed by any job or is followed by at least one 1-job on each machine.*

Proof. Let k and ℓ be two consecutive 2-jobs and assume there is at least one in-between 1-job on machine 1 and none on machine 2. Let t_1, t_2 be the start times of job k on respectively machine 1 and 2. We may assume without loss of generality that $t_1 \geq t_2$: otherwise we just swap the schedules of the two machines on the interval $[0, C_k]$ and then this inequality will hold. We change the schedule for k and ℓ and the in-between 1-jobs as follows. Job k is completely processed on machine 2, starting from time t_2, and the in-between 1-jobs from machines 1 are moved forward such that the first one starts at time t_1. Let x_{1k} be the amount of processing on job k that was previously assigned to machine 1, where we note that $x_{1k} \leq \frac{1}{2}p_k$. We increase the part of job ℓ on machine 1 by x_{1k}, and decrease the part of job ℓ on machine 2 by x_{1k}. This is possible, since the part of job ℓ that was previously on machine 2 is at least $\frac{1}{2}p_\ell \geq \frac{1}{2}p_k \geq x_{1k}$.

The completion time of each of the in-between 1-jobs decreases by $x_{1k}+s$, the completion time of job k increases by x_{1k} and the completion time of job ℓ remains unchanged. The total completion time is thus reduced by at least s. If job k is the last 2-job, followed only by 1-jobs on machine 1, we can make a similar adjustment and decrease the total completion time. □

Lemma 5. *In the case of two machines, there are no 1-jobs after a 2-job in an optimal schedule satisfying the properties of Lemmas 1, 2 and 3.*

Proof. Suppose the lemma is not true. Then there must be a 2-job h that is directly followed by a 1-job. By Lemma 4, there must be at least one such 1-job on each machine, say jobs k and ℓ. Assume without loss of generality that $p_k \leq p_\ell$. Let x_{1h} and x_{2h} be the processing time of h on machine 1 and 2, respectively. As argued before, without loss of generality we assume that $x_{1h} \geq x_{2h}$. Let us define the starting time of h as zero, and let $\Delta = x_{1h} - x_{2h}$. Note that $C_h = \frac{1}{2}(\Delta + p_h + 2s)$. Then, the sum of the three completion times is

$$\begin{aligned} C_h + C_k + C_\ell &= C_h + (C_h + p_k + s) + C_\ell \\ &= \Delta + p_h + 2s + p_k + s + C_\ell. \end{aligned} \tag{2.1}$$

We reschedule the jobs h, k, and ℓ as follows, while the remaining schedule stays the same. Place job h, the shortest among h, k and ℓ, on machine 1 (unsplit), job k on machine 2 (unsplit), and behind these two, job ℓ is split on machine 1 and 2, in such a way that it completes on one machine at time C_k and at time C_ℓ on the other. We denote by C'_h, C'_k and C'_ℓ the new completion times after this switch. The sum of these completion times becomes

$$C'_h + C'_k + C'_\ell = (p_h + s) + (\Delta + p_k + s) + C_\ell,$$

which is exactly s less than the sum of the three completion times in (2.1) from before the switch. □

Given the previous lemmas, we see that the 2-jobs are scheduled in SPT order at the end. By Lemma 2, the first 2-job, say job k, is not shorter than the preceding 1-jobs. But this implies that the 1-jobs can be scheduled in SPT order without increasing the completion time of job k or its following jobs. By considering each of the n jobs as the first 2-job, we immediately obtain a $O(n^2)$-time algorithm to solve the problem. Carefully updating consecutive solutions leads to a faster method.

Theorem 1. *There exists an $O(n \log n)$ algorithm for minimizing the total completion time of jobs on two identical parallel machines with job splitting and uniform setup times.*

Proof. Suppose we schedule the first k jobs (for any $1 \leq k \leq n$) in SPT order as 1-jobs and the other jobs in SPT order as 2-jobs. We would like to compute the change in objective value that results from turning job k from a 1-job into a 2-job. However, this happens to give a rather complicated formula. It is much easier to consider the change for job $k-1$ and k simultaneously.

The schedule for the 1-jobs $j < k-1$ does not change. To facilitate the exposition, suppose that job $k-1$ starts at time zero and job k starts at time t. Then $C_{k-1} + C_k = p_{k-1} + s + t + p_k + s$. After turning the jobs into 2-jobs, the new completion times become $C'_{k-1} = (t + p_{k-1} + 2s)/2$ and $C'_k = (t + p_{k-1} + p_k + 4s)/2$. Hence,

$$(C'_{k-1} + C'_k) - (C_{k-1} + C_k) = s - p_k/2.$$

In addition, each job $j > k$ completes s time units later. Hence, the total increase in objective value due to turning both job $k-1$ and k from a 1-job into a 2-job is

$$f(k) := (n - k + 1)s - p_k/2.$$

Notice that $f(k)$ is decreasing in k, since $s > 0$ and p_k is non-decreasing in k. Hence, either there exists some $k \in \{2, \ldots, n\}$ such that $f(k) < 0$ and $f(k-1) \geq 0$, or either $f(n) \geq 0$, or $f(2) < 0$.

Suppose there exists some $k \in \{2, \ldots, n\}$ such that $f(k) < 0$ and $f(k-1) \geq 0$. The optimal schedule is to have either $k-1$ or $k-2$ unsplit jobs, since the first inequality and monotonicity imply that a schedule with $k-2$ unsplit jobs has a better objective value than a schedule with k or more unsplit jobs, and the second inequality and monotonicity imply that a schedule with $k-1$ unsplit jobs has a better objective value than a schedule with $k-3$ or fewer unsplit jobs.

If $f(n) \geq 0$ then the optimal solution is either to have only 1-jobs or have only job n as a 2-job. If $f(2) < 0$ then the optimal solution is either to have only 2-jobs or have only job 1 as a 1-job.

Straightforward implementation of the above gives the desired algorithm, the running time of which is dominated by sorting the jobs in SPT order. □

2.5 Three and more machines

The properties exposed in Section 2.3 have been proven to hold for any number of machines. The properties presented in Section 2.4 were shown specifically for two machines only. In this section we investigate their analogues for three and more machines. We will present some examples of instances that show that the extension is far from trivial. It keeps the complexity of the problem on three and more machines as an intriguing open problem.

Lemma 5 shows that for two machines, there always exists an optimal schedule in which $|M_j|$ is monotonically non-decreasing in j (if the jobs are in SPT order). The following lemma shows that this does not hold for an arbitrary number of machines.

Lemma 6. *There exist instances for which there is no optimal schedule in which $|M_j|$ is monotonically non-decreasing in j.*

Proof. Consider the instance on three machines having 10 jobs with their vector of processing times $p = (3, 10, 10, 10, 10, 50, 50, 50, 50, 50)$ (1 small job, 4 medium-sized jobs and 5 large jobs) and $s = 0.7$. We slightly perturb the processing times if necessary, obtaining $p_j < p_{j+1}$ for all $j = 1, 2, \ldots, n-1$.

We found all optimal solutions for this instance by exhaustive search. An optimal solution is depicted in Figure 2.2(a). As we see, job 2 is split over machines 2 and 3, but job 3, starting later than job 2, is not split. Jobs 4 and 5 are again 2-jobs and are split over machines 2 and 3. The large jobs are all split over all three machines.

Below, we will describe all other optimal solutions to this instance. We will consider two solutions to be the same, if one solution can be obtained from the other by a relabeling of machines, and/or (repeatedly) swapping the schedule of two machines from some time t till the end of the schedule, if these two machines both complete processing of some job at time t.

The second optimal schedule, in Figure 2.2(b), is obtained by scheduling job 1 on machine 1, job 2 split on machines 2 and 3, job 3 on machine 2 (or 3), and jobs 4 and 5 as split jobs on the machines not used by job 3. The remaining jobs are again all split over

1	3		6	7	8	9	10
2	4	5	6	7	8	9	10
2	4	5	6	7	8	9	10

(a)

1	4	5	6	7	8	9	10
2	3		6	7	8	9	10
2	4	5	6	7	8	9	10

(b)

1	4		6	7	8	9	10
2	3	5	6	7	8	9	10
2	3	5	6	7	8	9	10

(c)

1	3	5	6	7	8	9	10
2	4		6	7	8	9	10
2	3	5	6	7	8	9	10

(d)

Figure 2.2: Gantt charts depicting the optimal solutions to the 3-machine instance with processing times $p = (3, 10, 10, 10, 10, 50, 50, 50, 50, 50)$ (1 small job, 4 medium-sized jobs and 5 large jobs) and $s = 0.7$.

all three machines. It is easily verified that the objective of this schedule is the same as the objective of the schedule in Figure 2.2(a): the completion time of job 3 increases by 2, and the completion times of jobs 4 and 5 each decrease by 1, and all other completion times remain the same. The remaining two optimal schedules, in Figures 2.2(c) and 2.2(d) are obtained by switching jobs 3 and 4 in the first two optimal schedules. We note that these schedules continue to be optimal if the processing times are slightly perturbed, as mentioned earlier.

All optimal solutions for this instance share the property that job 2 is a 2-job, and either job 3 or job 4 is a 1-job, which proves the lemma. □

If we slightly change the instance from the proof of Lemma 6 by deleting one of the large jobs, then there is a unique optimal solution, which splits job 3 over machines 1 and 2 and continues with splitting job 4 over machines 1 and 3. Job 5 and the four large jobs are split over all three machines, see Figure 2.3.

Lemma 6 and the fact that a subtle change in the problem instance causes such a substantial change in the optimal schedule bodes ill for an algorithmic approach like the one in Section 2.4.

Despite the negative result from Lemma 6, it is not the case that the properties in the lemmas from Section 2.4 do not hold at all for more than two machines.

Consider, for example, Lemma 4. The statement also holds for a setting with more than two machines, if there are two consecutive 2-jobs that are split over the same two machines i and i'. If only machine i would have at least one in-between 1-job, and machine i' none, the same interchange argument can be made. However, at this point it seems that observations like this are not sufficient to characterize an optimal solution for the case of more than two machines.

2.6 Approximation algorithm

We will now show a constant-factor approximation algorithm for our problem, for an arbitrary number of machines. We remark that we do not know whether this problem is NP-hard, but the examples in the previous section do show that the way a job is scheduled in an optimal schedule may depend on jobs that occur much later in the schedule. Our approximation algorithm, on the other hand, is remarkably simple, and only uses a job's processing time and the setup time to determine how to schedule the job.

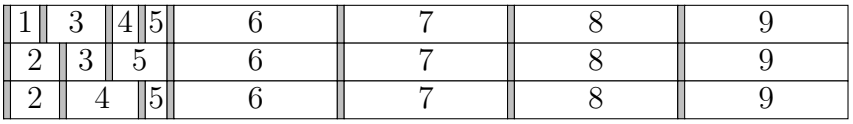

Figure 2.3: Gantt chart depicting the unique optimal solutions to the 3-machine instance with processing times $p = (3, 10, 10, 10, 10, 50, 50, 50, 50)$ (1 small job, 4 medium-sized jobs and 4 large jobs) and $s = 0.7$.

We schedule the jobs in order of non-decreasing processing times. Let $s > 0$ and let β be some constant that will be determined later. Job j will be scheduled such that it completes as early as possible under the restriction that it uses at most $g_j := \min\{\lceil \beta p_j/s \rceil, m\}$ machines. Thus, the job will be scheduled on the at most g_j machines that have minimum load in the schedule so far. It is easy to see that a job is always balanced this way.

Theorem 2. *The algorithm described above is a $(2+\beta)$-approximation algorithm for minimizing the total completion time with job splitting and uniform setup times, provided that $\beta \geq \frac{1}{4}(\sqrt{17}-1)$.*

Proof. Let $\mathcal{S}$ be the schedule produced by the described algorithm. Note that the total load (processing times plus setup times) of all jobs in $\mathcal{S}$ up to, but not including, job j is upper bounded by $L_j = \sum_{k<j}(p_k + g_k s)$, since job k introduced at most g_k setups. Therefore, the average load on the g_j least loaded machines is upper bounded by L_j/m. Since job j is balanced, we can thus upper bound the completion time $\tilde{C}_j$ of job j in the schedule by $L_j/m + p_j/g_j + s$. Note that this is an upper bound on the completion time of job j when we try to schedule it on at most g_j machines.

Noting that

$$\begin{aligned} p_j/g_j &= p_j/\min\{\lceil \beta p_j/s \rceil, m\} \\ &\leq p_j/\lceil \beta p_j/s \rceil + p_j/m \\ &\leq (1/\beta)s + p_j/m, \end{aligned}$$

and

$$g_k s = \min\{\lceil \beta p_k/s \rceil, m\} s < \beta p_k + s,$$

we obtain

$$\begin{aligned} \tilde{C}_j &\leq L_j/m + p_j/g_j + s \\ &\leq \frac{1}{m}\sum_{k<j}(p_k + g_k s) + p_j/g_j + s \\ &< \frac{1}{m}\sum_{k<j}\big((1+\beta)p_k + s\big) + p_j/m + (1+1/\beta)s \\ &\leq \frac{1+\beta}{m}\sum_{k\leq j} p_k + \left(\frac{j-1}{m} + 1 + \frac{1}{\beta}\right)s. \end{aligned}$$

Suppose we only needed a setup time for the first job to be processed on a machine, for any machine. Clearly, the optimal sum of completion times for this problem gives a lower bound on the sum of completion times in the optimum of the original problem. Now, the optimal schedule when we only need a setup time for the first job on a machine processes the jobs in SPT order and splits each job over all machines, which gives a sum of completion times of

$$\sum_j (s + \frac{1}{m}\sum_{k\leq j} p_k). \tag{2.2}$$

This gives one lower bound on the sum of completion times in an optimal schedule.

Further, in any schedule, at most m jobs are preceded by only one setup, at most another m by two setups, etc., giving a lower bound of $\sum_j \lceil j/m \rceil s$ on the sum of completion times; this is exactly the optimal value when all processing times are 0.

To bound this last lower bound in a more practical way, let $j = qm + a$ for some $q \geq 0$ and $a \in \{1, \ldots, m\}$. Then

$$\begin{aligned}\left\lceil \frac{j}{m} \right\rceil - \frac{j-1}{m} &= (q+1) - (qm + a - 1)/m \\ &= 1 - (a-1)/m.\end{aligned}$$

Now assume that $n = rm + b$, for some integer $r \geq 0$ and $b \in \{1, \ldots, m\}$. Then,

$$\begin{aligned}\sum_{j=1}^{n} \left\lceil \frac{j}{m} \right\rceil - \sum_{j=1}^{n} \frac{j-1}{m} &= r \sum_{a=1}^{m} \left(1 - \frac{a-1}{m}\right) + \sum_{a=1}^{b} \left(1 - \frac{a-1}{m}\right) \\ &= rm + b - r \sum_{a=1}^{m} \frac{a-1}{m} - \sum_{a=1}^{b} \frac{a-1}{m} \\ &= n - r(m-1)/2 - \frac{1}{2}(b-1)b/m \\ &\geq n - r(m-1)/2 - \frac{1}{2}(b-1) \\ &= n - (rm+b)/2 + r/2 + 1/2 \\ &= n/2 + r/2 + 1/2 \geq n/2.\end{aligned}$$

Hence, bringing the second summation to the right-hand side, and multiplying both sides with s yields

$$\sum_{j=1}^{n} \left\lceil \frac{j}{m} \right\rceil s \geq \sum_{j=1}^{n} \frac{j-1}{m} s + \frac{1}{2} ns. \tag{2.3}$$

Using $1 + \beta$ times lower bound (2.2), and one time lower bound (2.3), we get

$$\begin{aligned}(2+\beta) \sum_j C_j &\geq (1+\beta) \sum_j \left(s + \frac{1}{m} \sum_{k \leq j} p_k\right) + \left(\sum_j \frac{j-1}{m} s + \frac{1}{2} ns\right) \\ &= \sum_j \left(\frac{1+\beta}{m} \sum_{k \leq j} p_k + (1+\beta)s + \frac{j-1}{m} s + \frac{1}{2} s\right),\end{aligned}$$

which is at least as large as $\sum_j \tilde{C}_j$ provided $\beta > 0$ and $\frac{3}{2} + \beta \geq 1 + \frac{1}{\beta}$, which is equivalent to $\beta \geq \frac{1}{4}(\sqrt{17} - 1)$. □

Corollary 1. *There exists a* $2 + \frac{1}{4}(\sqrt{17} - 1) \approx 2.781$*-approximation algorithm for minimizing total completion time with job splitting and uniform setup times.*

2.7 Non-uniform setup times

The main part of this chapter considers uniform setup times. We use this term to denote that the setup times are job-, machine- and sequence-independent. In this section we will briefly reflect on how much of the results in this chapter remains valid if we consider either job-dependent or machine-dependent setup times.

2.7.1 Machine-dependent setup times

We will go over the lemmas of Sections 2.3 and 2.4 and see to what extent they hold for the case of machine-dependent setup times. It is very easy to see that the statements from Claim 1 and Lemma 1 remain valid, as they only consider a given solution per machine. And as the setup times *per machine* can be seen as uniform, the same proofs hold.

At first sight it seems that Lemma 2 should hold as well in the case of machine-dependent setup times. It is, however, not so obvious how to show this. A simple counter example shows that the proof given in Section 2.3 does not work for this case.

Consider jobs k and ℓ with $p_\ell = 3$, $p_k = 9$ and there is a machine (say machine 1) that schedules job k right before job ℓ. On machine 1, job k is processed for 2 time units and job ℓ is processed there completely. The remaining part of job k is processed on machine 2 and completes no later than the part on machine 1. On machine 1, the setup time $s_1 = 1$, while on machine 2, $s_2 = 3$. Following the proof of Lemma 2, we start by filling the machines in $M_k \backslash M_\ell$ (i.e., machine 2) with job ℓ. So we need a setup time of 3, then 3 time units for job ℓ. Then, there are 4 time slots remaining, to schedule another setup and 1 unit of job k. Now the remaining part of k (of size 8) will not fit in the slots left on machine 1, since after the setup time of 1, only 6 time units remain. The reason why this proof does not work out in the same way for the machine-dependent setup times, is that in the initial schedule we had two setups on machine 1 (of size 1, each) and one on machine 2 (of size 3) and after rescheduling, we have two setups on machine 2 and one on machine 1. Note that it is also not possible to simply switch the parts of job k and ℓ on machine 1, since then the new completion time of job ℓ, C'_ℓ, is strictly larger than C_k, while $C'_k = C_\ell$ and the objective function is worsened.

The intuition, however, still is that Lemma 2 should hold, even for machine-dependent setup times. Note that for the remaining lemmas, we are building upon schedules satisfying the previous lemmas, including Lemma 2. For the sake of argument, when discussing the remaining lemmas, we will assume that an optimal schedule exists satisfying the previous lemmas as stated.

For Lemma 3, given an optimal schedule with a minimum number of job parts, an LP is defined and it is shown that all constraints are tight, and hence all jobs are balanced. The constraints can be adjusted to incorporate machine-dependent setup times s_i without anything changing, so the claim of the lemma also holds.

Lemmas 4 and 5 are specifically proven for two machines only. In Lemma 4 it is assumed that k and ℓ are consecutive 2-jobs and only machine 1 has at least one 1-job in between and machine 2 none. By an interchanging argument it is shown that changing the order of these jobs improves the schedule by at least s. In the case of machine-dependent setup times it can be shown that the same interchanging of the jobs results in

an improvement of the schedule of at least s_1, the setup time on machine 1.

An interchange argument is also the basis for proving Lemma 5. Let us now assume that machine 1 starts no later setting up job h than machine 2 and define Δ as the difference between the moments that the machines start setting up for job h, i.e., $\Delta = (s_1 + x_{1h}) - (s_2 + x_{2h})$. Jobs k and ℓ are the 1-jobs following job h and we assume $p_k \leq p_\ell$. Then, repeating the argument from the proof of Lemma 5, it is not hard to see that interchanging the jobs leads to an improvement of s_i in the objective function, if job k is scheduled on machine i.

Note that even if all lemmas would hold, the construction of an optimal solution for the two-machine case is more difficult than in the case of uniform setup times. It still holds that we start with 1-jobs and from a certain job k we start splitting the jobs over two machines. But since the setup times on both machines are different, it could be the case that the leading 1-jobs are not alternately assigned to machines 1 and 2. For example, consider an instance of two machines where $s_1 = 1$ and $s_2 = 4$. We have jobs $1, 2, 3, 4$, with $p_j = j$. There are two optimal schedules. In the first one, jobs 1, 2 and 3 are scheduled on machine 1, and job 4 is scheduled on machine 2. In the other optimal schedule jobs 1, 2 and 4 are on machine 1 and job 3 is on machine 2. In both schedules the 1-jobs are not alternately assigned to the machines.

2.7.2 Job-dependent setup times

For job-dependent setup times, we will also consider all lemmas and see to what extent they hold.

In Claim 1, job parts that are on a machine are scheduled in a different order. Every job is split in exactly as many parts as before and hence the total amount of setup times is not changed. In the proof, we can define $y_{ij} = s_j + x_{ij}$ if $x_{ij} > 0$ and $y_{ij} = 0$ otherwise, and the proof still holds. This implies that also Lemma 1 is valid for this case.

Like in the case of machine-dependent setup times, also for job-dependent setup times the proof of Lemma 2 does not hold. Consider a job k and ℓ as in the proof. Thus, $p_\ell < p_k$, but there is a machine i processing job k right before job ℓ. Suppose machine i is the only machine in M_ℓ, whereas job k is split into many parts, all smaller than p_ℓ. And suppose $s_k < s_\ell$. Then, when rescheduling job ℓ into the slots of k in $M_k \backslash M_\ell$, job ℓ is split into at least two parts. Let job ℓ be split into π parts; then $\pi - 1$ extra setups of length s_ℓ are incurred. This then implies that job k saves $\pi - 1$ setups, but since their lengths s_k are strictly smaller than s_ℓ, we will not have enough space to reschedule k and ℓ in the way the proof dictates. In fact, it is not intuitively clear that this lemma would hold for job-dependent setup times.

In the proof of Lemma 3, we can adjust the constraints on the completion times such that $\sum_{k \leq j : M_k \ni i} (s_k + x_{ik}) \leq C_j$ for all j and all $i \in M_j$. Thus, this lemma also holds for job-dependent setup times.

The interchange argument of Lemma 4 can still be applied in the case of job-dependent setup times. It is easy to see that interchanging the 2-jobs and the in-between 1-job(s) gives an improvement of at least s_k, the setup time of job k.

In Lemma 5, however, the interchange argument is not necessarily holding. After the interchange, job h is set up one time less (i.e., from a 2-job it became a 1-job), and job

ℓ is set up once more (it became a 2-job from a 1-job). If $s_\ell > s_h$ it is not clear how the extra time that should be spent on setting up can be scheduled. Especially if there are other jobs after job ℓ, the interchange does not have to lead to an improved schedule.

2.8 Epilogue

In the following table we gather the state of the art on scheduling problems with job splitting and uniform setup times.

In the second column of the table, we summarize the complexity status of these problems. A question mark indicates that the complexity of the problem is unknown. In the third column we give the best approximation guarantee known, where a "-" indicates that no algorithm with a performance guarantee is known. If we consider it relevant, we also

Problem	Complexity	Algorithm
$P \mid \text{split} \mid \sum C_j$	in P	divide jobs equally over the machines in SPT order
$P2 \mid s, \text{split} \mid \sum C_j$	in P	algorithm of Section 2.4
$Pm \mid s, \text{split} \mid \sum C_j$	?	PTAS [63]
$P \mid s, \text{split} \mid \sum C_j$	?	2.781-approx. of Section 2.6
cf. $P \mid s, \text{pmtn} \mid \sum C_j$	in P	SPT
$P \mid \text{split} \mid \sum w_j C_j$	in P	divide jobs equally over the machines in WSPT order
cf. $P \mid \text{pmtn} \mid \sum w_j C_j$	NP-hard [24]	PTAS [1]
$P \mid s, \text{split} \mid \sum w_j C_j$	NP-hard [63]	-
cf. $P \mid s, \text{pmtn} \mid \sum w_j C_j$	NP-hard	-
$P \mid s, \text{split} \mid C_{\max}$	NP-hard [70] (cf. [28])	$\frac{5}{3}$-approximate split/ assignment [28]
$P \mid s, \text{split} \mid C_{\max}$ if $p_j \geq s\ \forall j$	NP-hard	$\frac{3}{2}$-approximate wrap-around algorithm [70]
cf. $P \mid s, \text{pmtn} \mid C_{\max}$	NP-hard [64]	PTAS [64]

Table 2.1: State of the art on scheduling problems with job splitting and setup times

present, as a footnote, the knowledge on the comparable version with preemption instead of splitting.

As can be seen from the table, the main open problem is the complexity of the scheduling problem we have considered in this chapter with an arbitrary number of machines. Already for 3 machines a complexity result would probably mean a breakthrough for the general case.

Another direction for future research is the weighted case. It was shown [63] that the problem of minimizing the weighted sum of completion times on parallel machines with job splitting and setup times is NP-hard. One possible direction would be to try adjusting the approximation algorithm from Section 2.6 to the weighted case. Or, for the weighted case on 2 machines, it could be considered to use parts of the approach of Sections 2.3 and 2.4 to obtain an approximation algorithm.

Note that apart from uniform setup times, there is a range of other interesting problems with job splitting and setup times that can be studied. As can be seen from Section 2.7, the setting with machine-dependent setup times is already more difficult than the setting with uniform setup times, but the setting with job-dependent setup times is even more complex. In the algorithm for two machines, we essentially split longer jobs, whereas the shorter jobs remain unsplit. With job-dependent setup times, however, it might be the case that a "longer" job has such a large setup time that it should never be split. This highly complicates any (approximation) algorithm for this problem.

Chapter 3

Assigning real-time tasks to unrelated machines

All results in this chapter appear in [58].

3.1 Introduction

This chapter considers scheduling of real-time tasks on *unrelated machines*. This model is motivated from the trend in hardware design towards heterogeneous processors. Modern hardware architectures often contain specialized processors for certain tasks (e.g., graphical processors, floating points units). To model the actual behavior of the different types of processors when making scheduling decisions, the unrelated-machines model is used. Therein it is assumed that the processing time of each task depends on the machine it is executed on, including the possibility that some tasks cannot be executed on some machines at all.

We search for an assignment of the tasks to the machines, such that all jobs released by a task are scheduled on the machine that the task was assigned to and all jobs meet their deadlines. Jobs are not allowed to migrate between machines. To the best of our knowledge, the unrelated-machines setting has not been studied for arbitrary-deadline sporadic tasks.

3.1.1 Related work for unrelated machines

The setting with unrelated machines is well-studied for the problem of assigning jobs to minimize the makespan. Lenstra, Shmoys and Tardos [53] give a 2-approximation algorithm for the problem of minimizing the makespan of a set of jobs and prove that it is NP-hard to achieve a performance ratio better than 1.5. Despite a lot of effort, the only improvements in the setting of an arbitrary number of machines are a 1.75-approximation algorithm for the graph balancing case [35] and a $33/17 \approx 1.94$-estimation algorithm for the restricted assignment case [69].

A generalization of this problem is the Generalized Assignment Problem (GAP). In this problem, assigning a job j to a machine i incurs a certain cost $c_{i,j}$. Shmoys and Tardos [65] give a 2 approximation for this problem; to be more precise, they devise an

algorithm computing a schedule with makespan $2T$ and cost at most C, given that a solution with the same cost and a makespan of T exists. See Section 3.3.1 for a more extensive description of this problem and the algorithm given by Shmoys and Tardos.

Azar and Epstein [8] consider ℓ_p norms (for finite $p > 1$, rather than the makespan) for which they improved the previously best result of $\theta(p)$ [7] to a 2 approximation and even a $\sqrt{2}$ approximation for the ℓ_2 norm. This was improved to a better-than-two result for all $p > 1$ in [50].

For the case of a constant number of machines, polynomial-time approximation schemes are known [45, 53]. Rather than focusing on a constant number of machines, one could also consider the case of a constant number of *machine types*; each machine belongs to one of the types and the processing time of each job depends only on the job and the type of the machine it is assigned to. For the problem of assigning implicit-deadline task systems to unrelated machines of two different types, an approximation algorithm is given in [61] based on the first-fit heuristic with a worst-case performance ratio of 2. For the same problem, a PTAS was given later [62]. In the meantime, a PTAS was given [72] for any constant number of machine types. The authors justify their contribution in [62] by pointing out that their PTAS for two different types has a much lower run-time complexity than the one in [72] applied to two machine types.

Note that when dealing with implicit-deadline task systems in the partitioned paradigm, the feasibility problem is equivalent to Makespan Minimization [41]. Hence, the results that hold for this latter problem can be applied to implicit-deadline task systems to minimize the maximum utilization over the processors.

3.1.2 Feasibility testing

The feasibility analysis of sporadic task systems on single processors has been extensively studied (see, for example, [19] for an overview). It is known that the Earliest Deadline First (EDF) algorithm, that schedules at any time the job with the earliest absolute deadline, is optimal in the sense that for any sequence of jobs it produces a valid schedule, whenever a valid schedule exists [56]. However, it is co-NP-hard to decide whether a task system is feasible on a single machine [37].

Recall Proposition 2 from Section 1.2.1 where the demand bound function specifies a necessary and sufficient condition for schedulability of a sporadic task system on a single processor. There are, however, exponentially many time points t to check. Albers and Slomka [2] suggested an approximate demand bound function, where for each task τ the number of test points of dbf_τ is limited. The approximation ratio then depends on this fixed number of test points; for time points beyond that fixed number, the demand bound function of each task is approximated by a linear function. Chen and Chakraborty [29] use the variant of this approximate demand bound function that, for each task τ, starts the linear approximation from the relative deadline d_τ. This approximate demand bound function gives a 2 approximation for the actual demand bound function and this factor is tight. It is, however, not tight in the resource augmentation setting. Chen and Chakraborty [29] show that the speedup required when using this approximate demand bound function for feasibility testing on a single processor is at most $\frac{2e-1}{e} \approx 1.632$.

On multiprocessor systems, there are two main paradigms for scheduling: *global* and

partitioned scheduling. In the former, all tasks can use all machines, and jobs can even be migrated from one machine to another. In the partitioned scheduling approach each task has to be assigned to one of the machines such that all its jobs have to be executed on this specific machine. Since the process of partitioning tasks among processors reduces a multiprocessor scheduling problem to a series of single-processor problems, the optimality of EDF for preemptive single-processor scheduling makes EDF a reasonable algorithm to use as the run-time scheduling algorithm on each machine. Note that the setting studied in this chapter follows the partitioned scheduling approach.

In the case of m identical processors, assuming the global paradigm, the natural EDF policy is no longer optimal, but it is known that any feasible collection of jobs on m machines of unit speed is schedulable using EDF on m machines of speed $2 - \frac{1}{m}$ [59]. Also, a corresponding test for task sets is known [16, 23]. Anand et al. [3] present an online algorithm needing only a speedup factor of $e/(e-1) \approx 1.58$.

In the case of the partitioned paradigm, let us first consider the special case of implicit deadlines. Then, a set of tasks is feasible on one machine if and only if the sum of the utilizations c_τ/p_τ is at most 1 and the problem reduces to Makespan Minimization. A PTAS is proposed for this problem where tasks are scheduled according to fixed priorities [36]. In that paper also the existence of an FPTAS is ruled out, thus showing that the problem is strongly NP-hard.

If deadlines are not implicit, much less is known. Baruah and Fisher [17] used the approximate demand bound function by Albers and Slomka [2] to provide the first non-trivial solution for partitioned scheduling of arbitrary- or constrained-deadline task systems on identical machines. They propose to use deadline-monotonic task partitioning; considering tasks iteratively from the shortest relative deadline, the algorithm assigns the task under consideration to a processor which is EDF schedulable when using the approximate demand bound function for the schedulability test. They prove that the deadline-monotonic partitioning has a $4 - \frac{2}{m}$ resource augmentation factor (where m is the number of processors) for arbitrary-deadline systems [17] and a $3 - \frac{1}{m}$ resource augmentation factor for constrained-deadline systems [18]. In [40] similar results are given if the tasks are scheduled according to static priorities, rather than with the more powerful EDF policy.

Chen and Chakraborty [29] improved the analysis of the deadline-monotonic policy with approximate demand bound functions and showed that it gives a feasibility test with speedup factor $\frac{3e-1}{e} - \frac{1}{m} \approx 2.6322 - \frac{1}{m}$ in case of constrained deadlines ($d_\tau \leq p_\tau$ for all τ) and a speedup factor of $3 - \frac{1}{m}$ for unconstrained deadlines.

3.1.3 Our results

To the best of our knowledge, no non-trivial algorithm is known for assigning a set of arbitrary-deadline sporadic tasks to a set of unrelated machines. We first present an algorithm that, given a task system for which a task assignment on m machines exists, finds a task assignment that can be scheduled on m machines that are $8 + 2\sqrt{6} \approx 12.9$ times as fast.

In order to obtain this result, one of the tools used is an LP rounding procedure that is presented separately in Section 3.3. From the fractional solution to an LP relaxation

it produces an integral solution that maintains an objective value not worse than that of the optimal fractional solution, while guaranteeing a bound on the violation of any constraint in the system. This procedure is based on ideas presented in [49], where a solution for an integer linear program (without objective function) is found by rounding the solution to the LP relaxation such that each variable is rounded up or rounded down and each constraint of the program is violated by at most a constant factor (depending on the maximum sum of elements over all columns of the coefficient matrix).

The rounding procedure that we developed is designed for rounding a fractional solution of the LP relaxation of any assignment-type problem where items (e.g., tasks in our case) have to be assigned to resources (e.g., machines) while obeying some knapsack-type constraints. Given that each resource-item combination appears in at most γ constraints, the procedure preserves the constraint that every item is assigned and all other constraints are violated by at most a factor of $\gamma + 1$. Further, the cost of this rounded solution is no more than the optimal cost of the LP relaxation. In fact, the well-known 2-approximation algorithm for the Generalized Assignment Problem by Shmoys and Tardos [65] can be derived as a direct corollary from our rounding procedure.

We apply the rounding procedure to a sparse LP formulation which approximately models the task assignment problem on unrelated machines and obtain the mentioned result (after some technical modifications of the LP). Also, we show that no polynomial-time algorithm can compute a task assignment needing a speedup factor of $2-\epsilon$ for $\epsilon > 0$, unless $P = NP$. Note that this bound is stronger than the best known $(3/2-\epsilon)$-hardness result for the contained problem of minimizing the makespan when scheduling jobs on unrelated machines [53].

For the case that the number of machines is fixed, we present a polynomial-time algorithm that either finds a feasible task assignment on m machines that are $1+\epsilon$ times as fast, or guarantees that no solution exists on unit-speed processors.

In order to be able to achieve these results, we need a deep understanding of the *demand bound function* (dbf) which yields a necessary and sufficient condition for a task system to be feasible on one machine. In particular, we present two new relaxations for handling this well-studied function. For our result for an arbitrary number of machines, we give a set of sparse linear constraints which approximate the dbf up to a constant factor. Due to the sparsity we are able to design the efficient iterative rounding procedure.

For the case of a constant number of machines, we observe that we cannot exploit the technique of partitioning the task set into "big" and "small" tasks as in the job scheduling problem. A task having a small execution time or small utilization might still be very tight in the sense that its relative deadline is fairly small. Therefore, even if all large tasks are already assigned, assigning the small tasks is still very tricky. An important feature of our dbf relaxation is that the feasibility test of assigning a task with deadline D to a machine having tasks with deadlines smaller than D already assigned to it, requires only limited information of the previously assigned tasks. To be more explicit, the approximate demand bound function for each task only needs to be evaluated at a constant number of points. Afterwards we just approximate the function by the task's utilization. We exploit this feature and other tricks to polynomially bound the running time of a dynamic programming algorithm.

3.2 Preliminaries

Given is a set M of m unrelated machines and a sporadic task system $\mathcal{T}$, with $|\mathcal{T}| = n$. Each task $\tau \in \mathcal{T}$ is characterized by a set of values $(\{c_{i,\tau}\}_{i\in M}, d_\tau, p_\tau)$, where $c_{i,\tau}$ is its execution time on machine i, d_τ is its deadline, and p_τ is the period. We assume all parameters to be integer and strictly positive. We study the problem of finding a task assignment $\mathcal{T} = \{\mathcal{T}_i\}_{i\in M}$ such that $\cup_i \mathcal{T}_i = \mathcal{T}$. Without loss of generality we can restrict to task assignments with the property that $\mathcal{T}_i \cap \mathcal{T}_{i'} = \emptyset$ for any two machines $i \neq i'$.

A task assignment is *feasible* if for any machine i, any job arrival sequence of the tasks in $\mathcal{T}_i$ can be feasibly scheduled on machine i, allowing jobs to be preempted. Since each task is assigned to exactly one machine, after finding an assignment, EDF will be our scheduling algorithm of choice, by its optimality for single-machine scheduling [56].

Recall that an *α-approximation test* for the problem of assigning tasks to unrelated machines is an algorithm that runs in polynomial time and either guarantees that there is no feasible integral assignment of the tasks to the given machines (running at unit speed), or finds an integral assignment which is feasible if the machines run at speed α.

By $u_{i,\tau}$ we denote the *utilization* of task τ on machine i and we define it as $u_{i,\tau} = c_{i,\tau}/p_\tau$. Given a task assignment $\mathcal{T}$, we define the utilization of each machine i by $u_i = \sum_{\tau\in\mathcal{T}_i} u_{i,\tau}$. Note that in a feasible assignment $u_i \leq 1$, for all $i \in M$, is a necessary but not sufficient condition for feasibility [56].

Clearly, the necessary condition $u_i \leq 1$ for all $i \in M$ implies that if there is a task τ such that $u_{i,\tau} > 1$, this task will never be assigned to machine i. Further, if for any task τ and machine i it holds that $c_{i,\tau} > d_\tau$, then τ will not be assigned to machine i. If it were assigned to machine i, the first job of τ clearly could not be completed before $c_{i,\tau}$ and would miss its deadline at d_τ.

We adjust Proposition 2 for the setting of unrelated machines.

Proposition 3 ([21])**.** *An assignment $\mathcal{T} = \{\mathcal{T}_i\}_{i\in M}$ is feasible for task system $\mathcal{T}$ if and only if for all $i \in M$*

$$dbf_{\mathcal{T},i}(t) := \sum_{\tau\in\mathcal{T}_i : d_\tau \leq t} \left\lfloor \frac{t + p_\tau - d_\tau}{p_\tau} \right\rfloor c_{i,\tau} \leq t \qquad \forall t \geq 0.$$

We write dbf_i instead of $dbf_{\mathcal{T},i}$ whenever the assignment $\mathcal{T}$ is clear from the context. Further, we define $dbf_i(\tau, t) := \lfloor (t + p_\tau - d_\tau)/p_\tau \rfloor\, c_{i,\tau}$, i.e., $dbf_i(\tau, t)$ denotes the contribution of task τ to $dbf_{\mathcal{T},i}(t)$.

For some integer k, we let denote $[k] = \{1, \ldots, k\}$ all integers from 1 until k.

3.3 Rounding procedure

Almost any combinatorial optimization problem can be formulated as an integer linear programming (ILP) problem with integral (or binary) decision variables and linear constraints. A feasible or optimal solution to this ILP is then a feasible or optimal solution to the problem originally considered. Unfortunately, solving ILPs in general is NP-hard [48]. A common approach is to, instead, solve the corresponding *LP relaxation*, where the integrality constraints on the decision variables are relaxed. In a second step, one turns

this fractional solution into an integral solution without (i) violating the constraints too much, or (ii) losing too much in the objective value with respect to the LP optimum.

An *iterative rounding procedure* is a procedure that step-by-step rounds and fixes a part (one or more variables) of a fractional solution, then computes a new fractional solution to the residual problem and again rounds some variables, etc. This is repeated until an integral solution is found for the original problem.

The interest in this type of methods was initiated by a breakthrough paper of Jain [44] in which he obtained a 2-approximate algorithm for a large class of minimum-cost network design problems in undirected networks. A later adaptation of his method by Singh and Lau [66] surprisingly involves no rounding at all. Only variables whose values are set to 1 in the linear programming relaxation are fixed and then the program is carefully updated. Indeed, it turned out that iterative methods can also be used to prove classic results in combinatorial optimization in a new way. Lau, Ravi and Singh [51] present in their monograph both new proofs for known results in exact optimization and approximation algorithms for constrained versions of those polynomially solvable problems.

In this section we present an LP rounding procedure that is specially designed for ILPs that stem from assignment problems. It guarantees that after applying the rounding, the obtained integral assignment violates the constraints from the ILP by only a bounded factor (depending on the structure of the LP matrix) and the objective value of the LP optimum is preserved. In Section 3.4 we use this procedure to obtain a constant-factor approximation test for the problem of assigning sporadic tasks to unrelated machines.

Since the procedure itself is independent of the application we will use it for, we present it here in more general terminology than we will need later. The rounding procedure is quite similar to the procedure presented in [49] by Karp et al., but in contrast to the latter, ensures that in the resulting integral solution one obtains a feasible integral assignment. A more detailed description of the procedure given by Karp et al. is given in Section 3.3.2.

Assume we want to solve the following assignment problem: given is a set of n items and a set of m resources M and a set $\Omega \subseteq \{1, \ldots, m\} \times \{1, \ldots, n\}$ containing all combinations (i, j) such that it is allowed to assign item j to resource i. For each combination $(i, j) \in \Omega$ we are given a cost $c_{i,j}$ denoting the cost of assigning item j to i. The goal is to assign each item j to a resource i to minimize the total cost, and such that for each resource i a set of knapsack-type constraints is satisfied. Each item appears in at most γ constraints per resource. Without loss of generality we assume that each item-resource combination appears in *exactly* γ of these constraints.

Given the set Ω, for all $(i, j) \in \Omega$ we define decision variable $z_{i,j}$, which is equal to 1 if item j is assigned to resource i, and 0 otherwise. Let $\theta := n\,\gamma$. We are also given non-negative values $a^{\ell}_{i,j}$ and b^{ℓ}_{i} for all $(i, j) \in \Omega$ and $\ell \in \{1, \ldots, \theta\}$, such that for each $(i, j) \in \Omega$, $a^{\ell}_{i,j} > 0$ for at most γ out of the θ coefficients. We want to solve the following integer linear program (ILP).

$$\min \sum_{(i,j)\in\Omega} c_{i,j} z_{i,j}$$

$$\text{s.t.} \sum_{i:(i,j)\in\Omega} z_{i,j} = 1 \qquad \text{for } j = 1,\dots,n; \tag{3.1a}$$

$$\sum_{j:(i,j)\in\Omega} a^{\ell}_{i,j} z_{i,j} \le b^{\ell}_{i} \qquad \text{for } i = 1,\dots,m,\ \ell = 1,\dots,\theta; \tag{3.1b}$$

$$z_{i,j} \in \{0,1\} \qquad \text{for all } (i,j) \in \Omega. \tag{3.1c}$$

We denote by LP^0 the LP relaxation of the above ILP. The contribution of item j in constraint ℓ of resource i is expressed by coefficient $a^{\ell}_{i,j}$. Note that if this is larger than the "capacity" b^{ℓ}_{i}, item j will never be assigned to resource i in an integral solution. However, for our reasoning in Section 3.4 we need to round LP relaxations of the above type where possibly $a^{\ell}_{i,j} > b^{\ell}_{i}$. We assume w.l.o.g. that $a^{\ell}_{i,j} \le 1$ for all i, j, ℓ (which can be ensured by linear scaling).

Solving the LP relaxation of this problem usually does not give an integral solution and hence items are fractionally assigned to multiple resources. Our rounding procedure turns this solution into an integral solution, such that each capacity constraint is not violated by more than γ times the maximum coefficient on the left-hand side. Further, the cost of this rounded solution will not be more than the optimal cost from the fractional LP solution. In typical assignment-type problems, the number of constraints where the same resource-item combination appears, is often low. For example, in the linear program that we will use in Section 3.4 for the problem of assigning tasks to unrelated machines, we will have that $\gamma = 2$.

3.3.1 Generalized Assignment Problem

The Generalized Assignment Problem (GAP), as it was first presented by Shmoys and Tardos [65], is defined as follows. Given are a set of n jobs and a set of m unrelated machines, for each combination of a job j and a machine i we are given the running time $p_{i,j}$ of job j on machine i and a cost $c_{i,j}$. For given values $C \ge 0$ and $T \ge 0$, the goal is to find an assignment of the jobs to the machines with total cost at most C and such that each machine has a makespan of at most T. The canonical LP formulation for GAP is the following:

$$\min \sum_{i=1}^{m}\sum_{j=1}^{n} c_{i,j} x_{i,j}$$

$$\text{s.t.} \sum_{i=1}^{m} x_{i,j} = 1 \qquad \text{for } j = 1,\dots,n; \tag{3.2a}$$

$$\sum_{j=1}^{n} p_{i,j} x_{i,j} \le T \qquad \text{for } i = 1,\dots,m; \tag{3.2b}$$

$$x_{i,j} \ge 0 \qquad \text{for all } (i,j) \text{ s.t. } p_{i,j} \le T. \tag{3.2c}$$

After solving this LP, Shmoys and Tardos [65] construct a bipartite graph where the vertices on one side represent the jobs, and the vertices on the other side represent the machines. Edges in this graph correspond to machine-job pairs with $x_{i,j} > 0$ and some cost structure on the edges is defined. Subsequently, a minimum-cost integer matching is found and each job is assigned to the machine that it is matched to in this matching. This polynomial-time algorithm finds a solution with cost at most C and makespan at most $2T$, given that a solution of cost at most C and makespan at most T exists.

Note that the LP formulation (3.2) fits nicely in the more general LP formulation given in (3.1). In fact, our iterative rounding procedure provides the same approximation bound and hence generalizes the result of Shmoys and Tardos. In (3.2), there is exactly one constraint per job-machine pair of the "capacity type" in (3.2b). Hence, when applying Theorem 4, that will follow in Section 3.3.3, the value of γ equals 1 and, as mentioned above, each constraint in (3.2b) will be violated by at most γ times the maximum coefficient on the left-hand side, which is at most T. Hence, our algorithm also gives a solution with cost at most C and makespan at most $2T$, given that a solution of cost at most C and makespan at most T exists. In fact, for this specific LP it is known that the iterative rounding procedure described above gives the mentioned result, see, e.g., [51].

3.3.2 Iterative rounding procedure by Karp et al.

As mentioned earlier in this chapter, the rounding procedure that was newly developed for the assignment of sporadic tasks to unrelated machines, is based on ideas from Karp et al. [49]. They consider the problem of routing wires on a VLSI chip and present as a tool a rounding algorithm for obtaining integral approximations to solutions of linear equations. They give the following Rounding Theorem.

Theorem 3 ([49])**.** *Let* $\mathbf{A}$ *be a real-valued* $r \times s$ *matrix, let* $\mathbf{x}$ *be a real-valued* s*-vector, let* $\mathbf{b}$ *be a real-valued* r*-vector such that* $\mathbf{Ax} = \mathbf{b}$*, and let* Δ *be a positive real number such that in every column of* $\mathbf{A}$*,*

(i) the sum of the positive elements is $\leq \Delta$*, and*
(ii) the sum of the negative elements is $\geq -\Delta$*.*

Then we can compute an integral s*-vector* $\hat{\mathbf{x}}$ *such that*

(i) for all i*,* $1 \leq i \leq s$*, either* $\hat{\mathbf{x}}_i = \lfloor \mathbf{x}_i \rfloor$ *or* $\hat{\mathbf{x}}_i = \lceil \mathbf{x}_i \rceil$ *(i.e.,* $\hat{\mathbf{x}}$ *is a "rounded version" of* $\mathbf{x}$*), and*
(ii) $\mathbf{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}$*, where* $\hat{\mathbf{b}}_i - \mathbf{b}_i < \Delta$ *for* $1 \leq i \leq r$*.*

The approach is as follows. In each step of the algorithm, either one of the variables is rounded up or down (if the number of constraints r is smaller than the number of variables s), or a constraint is removed (if $s \leq r$) and then the remaining program is re-solved. When $s \leq r$, constraint i will be picked to remove if it can be guaranteed that $\mathbf{A}_i\mathbf{w} < \mathbf{A}_i\mathbf{x} + \Delta$ for all integral solutions $\mathbf{w}$, such that, no matter how the final rounding will be done, $\hat{\mathbf{b}}_i - \mathbf{b}_i < \Delta$. Hence, this constraint can be dropped from explicit consideration.

Our own rounding procedure is quite similar to this one, but there is one important difference. In the LP formulation given in (3.1) we distinguish between the constraints of type (3.1a), assuring that each job is assigned to a machine, and the constraints of type (3.1b), regarding some capacity per machine. If we would apply the rounding procedure by Karp et al. to this LP, we would not be able to guarantee that the constraints in (3.1a) are always satisfied exactly. Our new procedure will guarantee this, at a small cost compared to the violation bound given by Karp et al. The next section describes our algorithm and proves its performance.

3.3.3 Our rounding procedure

Our rounding procedure is also iterative and in each iteration h we compute an extreme-point solution $\mathbf{z}^h$ of a linear program LP^h, where, as defined earlier, LP^0 equals the LP relaxation of the original assignment problem (3.1) and for $h \geq 0$ each LP^{h+1} is obtained by fixing the value for some variable(s) or removing some constraint(s) of LP^h. Whenever during this process constraints have become redundant (e.g., because all their variables have been already fixed), we remove them from the LP.

Given a feasible fractional solution $\mathbf{z}^h$, to obtain LP^{h+1} we first fix all variables which are integral in $\mathbf{z}^h$, i.e., those variables are not allowed to be changed anymore in the remainder of the procedure. Let s be the number of variables in LP^h and let r_a and r_b be the number of constraints of types (3.1a) and (3.1b), respectively. Let $r = r_a + r_b$. To obtain LP^{h+1}, we either fix and delete one or more variables (in case that $s > r$), or delete a constraint while ensuring that in the final solution that constraint will not be

ALGORITHM 1: Iterative rounding procedure

Input: A set of m resources, a set of n items and a set $\Omega \subseteq \{1, \ldots, m\} \times \{1, \ldots, n\}$ and for each $(i, j) \in \Omega$ and $\ell \in \{1, \ldots, \theta\}$ values $c_{i,j} \geq 0$, $b_i^\ell \geq 0$ and $a_{i,j}^\ell \geq 0$ and LP^0, the LP relaxation of (3.1)

loop

Solve LP^h consisting of s variables and r constraints and find extreme-point solution $\mathbf{z}^h$.

if $s > r$ **then**

$\mathbf{z}^h$ has at least one integral entry.

fix all integral entries at their value, remove them from the program and update right-hand side coefficients

remove all inequalities that have become redundant by fixing the integral values

else

find a resource i and $\ell \in \{1, \ldots, \theta\}$ such that $\max_{\mathbf{w} \in W} \sum_{j=1}^{n} a_{i,j}^\ell (w_{i,j} - z_{i,j}^h) \leq \gamma$, where $W = \{0, 1\}^s$

remove the inequality corresponding to the found i and ℓ.

end if

The remaining linear program is LP^{h+1}

end loop

return Integral solution $\hat{z}_{i,j}$ for all $(i, j) \in \Omega$ such that $c^\top \hat{\mathbf{z}} \leq c^\top \mathbf{z}^0$ and (3.6) holds.

violated too much. Later in this section we will explain how to identify these constraints and bound the amount that these constraints can be violated in the final solution. Along the way we ensure that the constraints of type (3.1a) are always satisfied exactly and that the costs in the objective are not increased.

Note that if there is some variable $z_{i,j}$ that is fixed at value 1 and removed from the program, for all $i' \in M \backslash \{i\}$, $z_{i',j}$ will be set to 0 and also be removed from the program. The constraint of type (3.1a) corresponding to this item j is then superfluous and will also be removed. We also update the right-hand side of each constraint where $z_{i,j}$ appears, taking into account that we decided to assign j to i which thus reduces the remaining capacities of i, for each of the up to θ remaining constraints.

The following lemma is the key to show that this algorithm works correctly.

Lemma 7. *Let LP^h be the linear program that is solved in iteration h of the rounding procedure, having s variables and r constraints. Let $\mathbf{z}^h$ be an extreme-point solution to this LP. If $s > r$, then $\mathbf{z}^h$ has at least one integral entry. If $s \leq r$, then there is a resource i and some $\ell \in \{1, \ldots, \theta\}$ such that*

$$\max_{\mathbf{w} \in W} \sum_{j=1}^{n} a_{i,j}^{\ell}(w_{i,j} - z_{i,j}^{h}) \leq \gamma,$$

where W is the integer solution space for all remaining variables, i.e., $W = \{0,1\}^s$.

Proof. Assume that the subproblem in iteration h (described by LP^h) is defined as $\mathbf{Az} \leq \mathbf{b}$. First assume that $s > r$. Then the null space of $\mathbf{A}$ is non-empty. Let $\mathbf{z}_0$ be a vector in the null space of $\mathbf{A}$. Since $\mathbf{z}^h$ is an extreme-point solution to LP^h it cannot be expressed as the convex combination of two (or more) solutions to LP^h (see also, e.g., [47]). If $\mathbf{z}^h$ would not have any integral entry then we could find a value $\delta > 0$ such that $\mathbf{z}^h + \delta\mathbf{z}_0$ and $\mathbf{z}^h - \delta\mathbf{z}_0$ were both solutions to LP^h and, in particular, $\mathbf{z}^h$ would be a convex combination of these two solutions. Therefore, $\mathbf{z}^h$ must have at least one integral entry.

Now assume that $s \leq r$. For this case, we show that there always exists a constraint ℓ of type (3.1b) such that $\max_{\mathbf{w} \in W}\{(\mathbf{Aw})_\ell - (\mathbf{Az})_\ell\} \leq \gamma$. We show the statement by contradiction. Assume that the statement is not true, that is, for each constraint ℓ of type (3.1b) it holds that there exists a vector $\mathbf{w} \in W$ such that

$$(\mathbf{Aw})_\ell - (\mathbf{Az})_\ell > \gamma. \tag{3.3}$$

In each previous round, if variables were removed from the program, also constraints that had become redundant, were removed. Therefore, for all variables present in the linear program in this round, the corresponding constraint of type (3.1a) is also still present in the linear program (and this constraint is not present if all of its variables have been removed from the program). It follows that

$$\sum_{i=1}^{m} \sum_{\substack{j : z_{i,j} \\ \text{remaining}}} z_{i,j} = r_a. \tag{3.4}$$

Define Λ as the set of constraints of type (3.1b) present in the current linear program. For any $\pi = (i, j)$, let Λ^π denote the set of these constraints containing variable z_π. Then,

$$\begin{aligned}
\gamma(r - r_a) &= \gamma\, r_b \\
&\overset{(3.3)}{<} \sum_{\ell \in \Lambda} \max_{\mathbf{w} \in W} ((\mathbf{A}\mathbf{w})_\ell - (\mathbf{A}\mathbf{z})_\ell) \\
&\overset{a_{\ell,\pi} \geq 0}{=} \sum_{\ell \in \Lambda} ((\mathbf{A}\mathbf{1})_\ell - (\mathbf{A}\mathbf{z})_\ell) \\
&= \sum_{\ell \in \Lambda} \sum_{\pi} a_{\ell,\pi}(1 - z_\pi) \\
&= \sum_{\pi} \sum_{\ell \in \Lambda^\pi} a_{\ell,\pi}(1 - z_\pi) \\
&\leq \sum_{\pi} \gamma(1 - z_\pi) \\
&= \gamma s - \sum_{\pi} \gamma z_\pi \\
&= \gamma(s - r_a). \qquad (3.5)
\end{aligned}$$

The second inequality follows since each variable z_π appears in at most γ constraints of type (3.1b) and since we assumed that $a_{\ell,\pi} \leq 1$ for all constraints $\ell \in \Lambda$ and all variables π.

The chain of inequalities implies that $\gamma(r - r_a) < \gamma(s - r_a) \Rightarrow r < s$ which is a contradiction to being in the case that $s \leq r$. Hence, we conclude that if $s \leq r$, there must be a constraint ℓ of type (3.1b), for which $\max_{\mathbf{w} \in W}\{(\mathbf{A}\mathbf{w})_\ell - (\mathbf{A}\mathbf{z})_\ell\} \leq \gamma$. □

Given an extreme-point solution $\mathbf{z}^h$ for a linear program LP^h, we can find in polynomial time a constraint which can be dropped, assuming that the number of variables is bounded by the number of constraints (i.e., $s \leq r$). For each of the polynomial number of constraints, we try the most "pessimistic" vector $\mathbf{w} \in W = \{0, 1\}^s$. This vector $\mathbf{w}$ is obtained by taking $w_{i,j} = 1$ for all (i, j), since $a^\ell_{i,j} \geq 0$ for all (i, j) and all $\ell \in \{1, \ldots, \theta\}$.

Theorem 4. *Let $m, n, \gamma \in \mathbb{N}$. Let $\theta = n\,\gamma$. Suppose we are given a set $\Omega \subseteq \{1, \ldots, m\} \times \{1, \ldots, n\}$ of pairs (i, j) that we define a decision variable $z_{i,j}$ for. For each $(i, j) \in \Omega$ and $\ell \in \{1, \ldots, \theta\}$ we are given values $c_{i,j} \geq 0$, $b^\ell_i \geq 0$, and $a^\ell_{i,j} \geq 0$, such that for each $(i, j) \in \Omega$, $a^\ell_{i,j} > 0$ for at most γ coefficients. Assume that the resulting linear program LP^0, as given in (3.1), is feasible and denote by $\mathbf{z}^*$ its optimal solution. Then there is a polynomial-time algorithm computing an integral solution $\hat{\mathbf{z}}$ with $c^\top \hat{\mathbf{z}} \leq c^\top \mathbf{z}^*$ which satisfies*

$$\sum_{i:(i,j) \in \Omega} \hat{z}_{i,j} = 1 \qquad \text{for } j = 1, \ldots, n; \qquad (3.6a)$$

$$\sum_{j:(i,j) \in \Omega} a^\ell_{i,j} \hat{z}_{i,j} \leq b^\ell_i + \gamma \max_j a^\ell_{i,j} \qquad \text{for } i = 1, \ldots, m,\ \ell = 1, \ldots, \theta; \qquad (3.6b)$$

$$\hat{z}_{i,j} \in \{0, 1\} \qquad \text{for all } (i, j) \in \Omega. \qquad (3.6c)$$

Proof. Before applying the rounding procedure, we scale each linear constraint such that $\max_j a^\ell_{i,j} = 1$ for each combination of a resource i and a value $\ell \in \{1, \ldots, \theta\}$.

We apply our iterative rounding procedure, where LP^0 equals the LP relaxation of (3.1) and for each $h \geq 0$, LP^{h+1} is obtained from LP^h by either fixing the value of some variable(s) or removing some constraint(s). Suppose we computed an extreme-point solution $\mathbf{z}^h$ for LP^h. Then if $s > r$, according to Lemma 7, $\mathbf{z}^h$ must have at least one integral value. Those variables that have an integer value in $\mathbf{z}^h$ are fixed at these values and the variables are removed from the program. If a variable $z_{i,j}$ is fixed at value 1, then for all $i' \in M \backslash \{i\}$, the variables $z_{i',j}$ will be fixed at value 0 and be removed and the constraint of type (3.1a) corresponding to j will also be removed. Also the right-hand side of each constraint of type (3.1b) where $z_{i,j}$ appears will be updated, considering that assigning item j to resource i used $a^\ell_{i,j}$ of the capacity in constraint ℓ. Note that the optimal objective value of the resulting linear program LP^{h+1} is not larger than the optimal objective in LP^h since $\mathbf{z}^h$ induces a solution to LP^{h+1} with the same objective value as $\mathbf{z}^h$ gives in LP^h.

If $s \leq r$, we know by Lemma 7 that a constraint ℓ exists such that

$$\max_{\mathbf{w} \in W} \{(\mathbf{A}\mathbf{w})_\ell - (\mathbf{A}\mathbf{z})_\ell\} \leq \gamma \tag{3.7}$$

and it can be found in polynomial time. To this end, we check for each resource i whether $\sum_j (1 - z_{i,j}) \leq \gamma$. This is sufficient since all $a^\ell_{i,j} \geq 0$ and the maximum value any variable $z_{i,j}$ can take is 1. (In fact, it is even true that if the latter condition is satisfied *all* constraints of a given resource can be removed.) Since the removed constraint satisfies (3.7), we know that whatever value the variables in this constraint will get in the final solution, the additive error in the right-hand side of this constraint will be at most γ, assuming that all $a^\ell_{i,j} \leq 1$. Note that removing constraints cannot increase the costs of the final solution. At the end we scale all constraints back to their original values. Thus, the resulting solution satisfies (3.6b).

Finally, we know that in each iteration either of the two cases of Lemma 7 applies and thus after a polynomial number of iterations either all constraints or all variables are removed and we are done. □

3.4 Arbitrary number of machines

In this section we present an $8 + 2\sqrt{6} \approx 12.9$-approximation test for assigning tasks to unrelated machines and we show that the problem is NP-hard to approximate to within a ratio of $2 - \epsilon$ for any $\epsilon > 0$.

3.4.1 Approximate demand bound function

As indicated earlier, the *demand bound function* gives a sufficient and necessary condition for feasibility of a task system on a single processor [21] (see also Proposition 2 in Section 1.2.1). It leads to an exponential-time test, however. This has led to the development of approximate demand bound functions.

Albers and Slomka [2] propose an approximate demand bound function that after a constant number of evaluations of the dbf, approximates it by a linear function. Starting the linear approximation already at $t = d_\tau$, the following approximate dbf is obtained.

$$dbf^*(\tau, t) = \begin{cases} 0 & \text{if } t < d_\tau, \\ \left(\frac{t-d_\tau}{p_\tau} + 1\right) c_\tau & \text{otherwise.} \end{cases}$$

This is a 2 approximation for the real demand bound function [29].

In the next subsection, we give an LP formulation that approximates the demand bound function with a higher approximation ratio, which leads to the question why the approximate dbf given here was not used. The answer to that question lies in the rounding procedure presented in Section 3.3.3. The linear inequalities that we use in our LP formulation have the property that each task-machine combination appears in only two constraints. Recall that the number of times each combination appears in constraints of type (3.1b) has a direct influence on the bound on the violation of each of the constraints when applying the rounding procedure. Using the above approximate dbf of Albers and Slomka in an LP formulation would yield up to n appearances of a task-machine combination in the "capacity"-type constraints. Hence, this would yield a much worse approximation ratio for the overall problem.

3.4.2 Constant-factor approximation test

Relaxed dbf constraints

We will formulate the problem of assigning tasks to unrelated machines as a linear program, such that the tasks on each machine can be feasibly scheduled using the EDF-scheduler. First, we derive a set of linear inequalities which are

- necessary, meaning that they are fulfilled by any feasible assignment,
- approximately sufficient, meaning that any integral assignment which (approximately) fulfills the constraints is feasible if the speed of the machine is increased by some constant factor, and
- sparse, meaning that each variable occurs in only two capacity constraints.

For each pair of a machine i and a task τ, such that $u_{i,\tau} \leq 1$ and $c_{i,\tau} \leq d_\tau$, we introduce a variable $y_{i,\tau}$ modeling to assign τ to machine i. Note that for pairs (i, τ) that do not satisfy these conditions we do not need a variable $y_{i,\tau}$ since it will be infeasible to assign such a τ to machine i. The first constraints are utilization bounds on all tasks assigned to the same machine i. We demand that

$$\sum_{\tau \in \mathcal{T}} u_{i,\tau} y_{i,\tau} \leq 1 \qquad \forall i \in M. \tag{3.8}$$

Secondly, we require that for all tasks with their deadline in the interval $(2^{k-1}, 2^k]$, the sum of their execution time is at most 2^k. This gives

$$\sum_{\tau \in \mathcal{T} : d_\tau \in (2^{k-1}, 2^k]} c_{i,\tau} y_{i,\tau} \leq 2^k \qquad \forall i \in M, \forall k \in \mathbb{N}. \tag{3.9}$$

We call these conditions the *relaxed dbf constraints*. It is clear that these constraints have to be fulfilled by any feasible task assignment. Since they are linear, they can be used in an LP relaxation for the problem. Their sparsity gives the potential to use the rounding procedure from Section 3.3 to obtain an integral solution that violates the relaxed dbf constraints only by a constant factor. The following lemma shows that—even when violated up to a constant factor–the presented constraints are approximately sufficient.

Lemma 8. *Let $\mathcal{T}$ be an assignment for the task system $\mathcal{T}$ such that, for all machines i, $\sum_{\tau\in\mathcal{T}_i} u_{i,\tau} \leq \beta$ and $\sum_{\tau\in\mathcal{T}_i : d_\tau \in (2^{k-1},2^k]} c_{i,\tau} \leq \beta\, 2^k$. Then $dbf_{\mathcal{T},i}(t) \leq 6\beta t$ for all $t \geq 0$ and $\mathcal{T}$ is a feasible task assignment under a speedup factor of 6β.*

Proof. Let $q \in \mathbb{N}$ and $t := 2^q$. Consider an assignment $\mathcal{T}$ of the tasks in $\mathcal{T}$ to the machines in M. For any machine $i \in M$, we bound $dbf_i(t) := dbf_{\mathcal{T},i}(t)$ by

$$\begin{aligned}
dbf_i(t) &= \sum_{\tau\in\mathcal{T}_i :\, d_\tau \leq t} \left\lfloor \frac{t + p_\tau - d_\tau}{p_\tau} \right\rfloor c_{i,\tau} \\
&\leq \sum_{\tau\in\mathcal{T}_i :\, d_\tau \leq t} \left(t\frac{c_{i,\tau}}{p_\tau} + c_{i,\tau} \right) \\
&\leq t \sum_{\tau\in\mathcal{T}_i} \frac{c_{i,\tau}}{p_\tau} + \sum_{\tau\in\mathcal{T}_i :\, d_\tau \leq t} c_{i,\tau} \\
&= t \sum_{\tau\in\mathcal{T}_i} u_{i,\tau} + \sum_{k=0}^{\log_2(t)} \sum_{\tau\in\mathcal{T}_i :\, d_\tau \in (2^{k-1},2^k]} c_{i,\tau} \\
&\leq \beta t + \sum_{k=0}^{q} \beta\, 2^k \\
&\leq \beta t + \beta\, 2^{q+1} \\
&= \beta t + 2\beta t = 3\beta t.
\end{aligned}$$

Hence, for any machine i and for arbitrary t we get that $dbf_i(t) \leq dbf_i(2^{\lceil \log_2 t \rceil}) \leq 3\beta\, 2^{\lceil \log_2 t \rceil} \leq 6\beta t$. This implies that $\mathcal{T}$ is a feasible assignment for scheduling the tasks in $\mathcal{T}$ to the set of unrelated machines whenever the machines receive a speedup factor of 6β. □

The approximation algorithm

Let $\rho > 1$. We define the function $r(x) := \rho^{\lceil \log_\rho x \rceil}$. Assume we are given an instance of our problem. Let $d_{\max} := \max_{\tau\in\mathcal{T}} d_\tau$ and define the set $\mathcal{D}_\rho := \{\rho^0, \rho^1, \ldots, r(d_{\max})\}$. We now formulate the following linear program, denoted by Ass-LP, where the deadlines are rounded up to the nearest power of ρ rather than to the nearest power of 2 (as implicitly

done in the relaxed dbf constraints given earlier).

$$\sum_{i \in M} y_{i,\tau} = 1 \qquad \forall \tau \in \mathcal{T}; \tag{3.10a}$$

$$\sum_{\tau \in \mathcal{T}} u_{i,\tau} y_{i,\tau} \leq 1 \qquad \forall i \in M; \tag{3.10b}$$

$$\sum_{\tau \in \mathcal{T} : r(d_\tau) = D} c_{i,\tau} y_{i,\tau} \leq D \qquad \forall D \in \mathcal{D}_\rho,\ \forall i \in M; \tag{3.10c}$$

$$y_{i,\tau} \geq 0 \qquad \forall \tau \in \mathcal{T},\ \forall i \in M : u_{i,\tau} \leq 1 \wedge c_{i,\tau} \leq d_\tau. \tag{3.10d}$$

If Ass-LP is infeasible, then there can be no feasible (integral) task assignment. Now assume that it is feasible and we have computed a feasible solution $\mathbf{y}^*$. For each machine i and deadline $D \in \mathcal{D}_\rho$ we extract a value

$$U_{i,D} := \sum_{\tau \in \mathcal{T} : r(d_\tau) = D} c_{i,\tau} y^*_{i,\tau}.$$

Based on these values, we define a strengthened variation of Ass-LP, denoted by SAss-LP in the sequel. We obtain the latter by replacing the constraints (3.10c) by the following set of constraints:

$$\sum_{\tau \in \mathcal{T} : r(d_\tau) = D} c_{i,\tau} y_{i,\tau} \leq U_{i,D} \qquad \forall D \in \mathcal{D}_\rho,\ \forall i \in M. \tag{3.10c'}$$

Clearly if $\mathbf{y}^*$ is a feasible solution for Ass-LP, it is also a feasible solution for SAss-LP and if SAss-LP is infeasible, then no feasible task assignment exists. We now round $\mathbf{y}^*$ to an integral solution which *approximately* satisfies SAss-LP by using the rounding procedure presented in Section 3.3. Note that each machine-task combination appears in one constraint of type (3.10b) and in one of type (3.10c'). Hence, when applying Theorem 4, we have that $\gamma = 2$. Therefore, applying the rounding procedure from Theorem 4 gives a task assignment $\hat{\mathbf{y}}$ that satisfies constraints (3.10a) and (3.10d) and the following two inequalities

$$\sum_{\tau \in \mathcal{T}} u_{i,\tau} \hat{y}_{i,\tau} \leq 3 \qquad \forall i \in M; \tag{3.11}$$

$$\sum_{\tau \in \mathcal{T} : r(d_\tau) = D} c_{i,\tau} \hat{y}_{i,\tau} \leq U_{i,D} + 2D \qquad \forall D \in \mathcal{D}_\rho,\ \forall i \in M. \tag{3.12}$$

Observe that $U_{i,D} \leq D$ for all $D \in \mathcal{D}_\rho$ and all machines i, and hence the vector $\hat{\mathbf{y}}$ violates the relaxed dbf constraints by at most a factor of $\beta = 3$. Hence, Lemma 8 directly implies that the task assignment given by the vector $\hat{\mathbf{y}}$ is feasible with a speedup of 18 if we choose $\rho = 2$. However, using the definition of $U_{i,D}$ and a more careful calculation, we can bound the needed speedup even further.

Lemma 9. *If we choose $\rho = 1 + \sqrt{6}/3$, the task assignment implied by the variables $\hat{\mathbf{y}}$ is feasible if the machines run at speed $8 + 2\sqrt{6}$.*

Proof. Let $\mathcal{T}$ denote the assignment implied by the integral variables $\hat{\mathbf{y}}$. Correspondingly, for all i, $\mathcal{T}_i$ denotes the set of tasks that are assigned to machine i, that is $\mathcal{T}_i = \{\tau \in \mathcal{T} : \hat{y}_{i,\tau} = 1\}$. We show that for the task assignment implied by $\hat{\mathbf{y}}$, $dbf_{\mathcal{T},i}(t) \leq (8 + 2\sqrt{6})t$ for all t and any machine i, if we choose $\rho = 1 + \sqrt{6}/3$:

$$\begin{aligned}
dbf_{\mathcal{T},i}(t) &= \sum_{\tau \in \mathcal{T}_i:\, d_\tau \leq t} \left\lfloor \frac{t + p_\tau - d_\tau}{p_\tau} \right\rfloor c_{i,\tau} \\
&\leq \sum_{\tau \in \mathcal{T}_i:\, r(d_\tau) \leq r(t)} \left(t \frac{c_{i,\tau}}{p_\tau} + c_{i,\tau} \right) \\
&\leq t \sum_{\tau \in \mathcal{T}} \frac{c_{i,\tau}}{p_\tau} \hat{y}_{i,\tau} + \sum_{\tau \in \mathcal{T}:\, r(d_\tau) \leq r(t)} c_{i,\tau}\, \hat{y}_{i,\tau} \\
&= t \sum_{\tau \in \mathcal{T}} u_{i,\tau} \hat{y}_{i,\tau} + \sum_{k=0}^{\log_\rho(r(t))} \sum_{\tau \in \mathcal{T}:\, r(d_\tau) = \rho^k} c_{i,\tau}\, \hat{y}_{i,\tau} \\
&\overset{(3.11)\&(3.12)}{\leq} 3t + \sum_{k=0}^{\log_\rho(r(t))} \left(U_{i,\rho^k} + 2\rho^k \right) \\
&\leq 3t + \rho^{\log_\rho(r(t))} + 2 \sum_{k=0}^{\log_\rho(r(t))} \rho^k \\
&= 3t + r(t) + 2\, \frac{\rho^{\log_\rho(r(t))+1} - 1}{\rho - 1} \\
&\leq 3t + r(t) + 2\rho \frac{r(t)}{\rho - 1} \\
&\leq t(3 + \rho + 2\frac{\rho^2}{\rho - 1}) = \left(8 + 2\sqrt{6}\right) t.
\end{aligned}$$

The last inequality follows since $r(t) \leq \rho t$, whereas the last equality follows by optimization of the value of ρ, i.e., by setting $\rho := 1 + \sqrt{6}/3$. □

Theorem 5. *There is a $(8 + 2\sqrt{6})$-approximation test for the problem of assigning tasks to unrelated machines.*

Proof. We solve the linear program Ass-LP and if feasible, we formulate the program SAss-LP given by (3.10a), (3.10b), (3.10c') and (3.10d). We then apply Theorem 4 to obtain the integral vector $\hat{\mathbf{y}}$, noting that $\gamma = 2$. The result follows from choosing $\rho = 1 + \sqrt{6}/3$ and applying Lemma 9. □

3.4.3 Hardness result

Finally, we show that it is NP-hard to decide whether a task system $\mathcal{T}$ has an assignment which is feasible on m unrelated machines, even with a speedup factor of $2 - \epsilon$, for any $\epsilon > 0$; the proof follows the lines of the $(\frac{3}{2} - \epsilon)$-hardness result for makespan minimization in [53]. We reduce from the 3-Dimensional Matching problem which is known to be NP-complete [53] and is defined below.

Instance of 3-Dimensional Matching: Given are three disjoint sets, consisting of n elements each: $E = \{e_1, e_2, \ldots, e_n\}$, $G = \{g_1, g_2, \ldots, g_n\}$ and $H = \{h_1, h_2, \ldots, h_n\}$. Also, given is a family $F = \{F_1, F_2, \ldots, F_m\}$ of $m \geq n$ triples such that each triple contains exactly one element from the set E, one from G, and one from H, that is, $|F_i \cap E| = |F_i \cap G| = |F_i \cap H| = 1$ for all $i \in \{1, \ldots, m\}$.
Question: Does F contain a 3-dimensional matching, i.e., a subfamily $F' \subseteq F$ for which $|F'| = n$ and $\bigcup_{F_i \in F'} F_i = E \cup G \cup H$?

Let $\epsilon > 0$. Consider an instance $\mathcal{I}$ of 3-Dimensional Matching. We create an instance (task system) $\mathcal{T}$ for sporadic real-time scheduling on unrelated machines in the following way:

- Define a large constant $M := 2/\epsilon$.
- Associate a machine i with each triple F_i, yielding m machines.
- Let $F(e_j) \subseteq F$ be the set of triples containing element $e_j \in E$. We will slightly abuse notation and also use $F(e_j)$ to refer to the set of machines corresponding to the triples in $F(e_j)$. Further, let $f(e_j) := |F(e_j)|$. Note that $\sum_{e_j \in E} f(e_j) = m$.
- For each element $g_k \in G$, create one task τ_k such that $d_{\tau_k} = 1$ and $p_{\tau_k} = \infty$. Further, $c_{i,\tau_k} = 1$ if $g_k \in F_i$, and $c_{i,\tau_k} = 2$, otherwise. We refer to these tasks as being type I tasks. There will be n tasks of type I.
- For each element $h_l \in H$, create one task τ_l such that $d_{\tau_l} = M$ and $p_{\tau_l} = \infty$. Further, $c_{i,\tau_l} = M - 1$ if $h_l \in F_i$, and $c_{i,\tau_l} = 2M$, otherwise. We refer to these tasks as being type II tasks. There will be n tasks of type II.
- For each element $e_j \in E$, create $f(e_j) - 1$ dummy tasks $dum(j_1), \ldots, dum(j_{f(e_j)-1})$ which have a deadline and period equal to 1. Each dummy task $dum(j_q)$ has $c_{i,j_q} = 1$ if $i \in F(e_j)$, and $c_{i,j_q} = 2$, otherwise. Note that in total there will be $m - n$ dummy tasks.

In Lemmas 10 and 11 we show that the reduction from 3-Dimensional Matching is valid.

Lemma 10. *If there exists a 3-dimensional matching for $\mathcal{I}$, then there exists a feasible assignment for $\mathcal{T}$.*

Proof. Let $F^* \subseteq F$ be the triples in the 3-dimensional matching. For all triples $F_i = \{e_j, g_k, h_l\} \in F^*$, schedule tasks τ_k and τ_l on machine i. Note that machine i can process the tasks assigned to it. Also, all tasks corresponding to elements in G and H have been scheduled. As the element e_j is only covered by one triple in F^*, it follows that there are $f(e_j) - 1$ machines remaining in $F(e_j)$ which are not assigned any tasks yet. Assign the $f(e_j) - 1$ dummy tasks $dum(j_1), \ldots, dum(j_{f(e_j)-1})$ to these machines. Note that a machine in $F(e_j)$ can process exactly one dummy task corresponding to the element e_j. □

Before giving Lemma 11, we first need to show some propositions about the elements in the created scheduling instance $\mathcal{T}$.

Proposition 4. *For any $\rho < 2$, no task τ_k corresponding to an element $g_k \in G$ can be scheduled on a machine i if $g_k \notin F_i$, even under a speedup of ρ.*

Proposition 5. *For any $\rho < 2$, no task τ_l corresponding to an element $h_l \in H$ can be scheduled on a machine i if $h_l \notin F_i$, even under a speedup of ρ.*

Proposition 6. *For any $\rho < 2$, no dummy task $dum(j_q)$ corresponding to an element $e_j \in E$ can be scheduled on a machine i if $i \notin F(e_j)$, even under a speedup of ρ.*

Proof. We argue for Proposition 4. Propositions 5 and 6 follow similarly. The proof is by contradiction; let a task τ_k corresponding to an element $g_k \in G$ be scheduled on a machine i such that $g_k \notin F_i$. Then $c_{i,\tau_k} = 2$. At time $d_{\tau_k} = 1$ the first job of task τ_k needs to be completed and hence a speedup of 2 is required. □

Proposition 7. *For any $\rho < 2 - \epsilon/2$, no dummy task $dum(j_q)$ can be scheduled together with another task on the same machine, even under a speedup of ρ.*

Proof. We argue by contradiction. We consider three cases:

- Suppose that two dummy tasks were scheduled on the same machine. Both dummies would need to finish their first job by their first deadline which is at time 1. Their accumulated processing requirement to the machine would be at least 2 and hence a speedup of at least 2 would be required.
- Suppose a dummy task and a task of type I were scheduled on the same machine. We reason analogously to the previous case.
- Suppose a dummy task and a task of type II were scheduled on the same machine. Consider time instant M, where the dummy task should have finished M jobs, whereas the other task should have finished its first job. The accumulated processing requirement for the machine by time M would be at least $M \cdot 1 + (M-1) = 2M - 1$. Hence, a speedup of $\frac{2M-1}{M} = 2 - \frac{1}{M} = 2 - \epsilon/2 > \rho$ would be required, by our definition of M. □

Lemma 11. *For any $\rho < 2 - \epsilon/2$, if there exists a feasible assignment for $\mathcal{T}$ with speedup ρ, then there exists a 3-dimensional matching for $\mathcal{I}$.*

Proof. Proposition 7 yields that each dummy task gets its own machine, even under a speedup of $\rho < 2 - \epsilon/2$. Therefore, and by Proposition 6, for all $e_j \in E$, there remains in each group $F(e_j)$ one machine available to process tasks of type I or II, even under a speedup of $\rho < 2$. In total there are $m - (m - n) = n$ machines left which do not process a dummy task. There are $2n$ tasks of type I and II. Since a single machine cannot process two tasks of the same type under a speedup less than 2, it follows that each machine which does not process a dummy task, processes one task of type I and one task of type II. Let $i_j \in F(e_j)$ be the machine which does not process a dummy task but instead one task of type I and one task of type II. By Propositions 4 and 5, the only way for machine i_j to be feasible, even under a speedup of $\rho < 2$, is when it processes the tasks corresponding to g_k and h_l where $F_{i_j} = \{e_j, g_k, h_l\}$. It follows that the machines i_j, for $j = 1, \ldots, m$, define a 3-dimensional matching for $\mathcal{I}$. □

Theorem 6. *Let $\epsilon > 0$. There is no $(2-\epsilon)$-approximation test for the problem of assigning tasks to unrelated machines, unless $P = NP$.*

Proof. Theorem 6 follows by Lemmas 10 and 11, and by the 3-Dimensional Matching problem being NP-complete. □

We remark that our hardness result is different from the one by Andersson and Tovar [4]. They show that any algorithm needs a speedup factor of at least $2-\epsilon$ for finding a feasible partition on m related parallel machines for a given task system with implicit deadlines in case the task system is feasible when migration is allowed.

3.5 Constant number of machines

Assuming that the number of machines m is a constant, in this section we present a polynomial-time dynamic-programming (DP) algorithm that gives a $(1+\epsilon)$-approximation test for any $\epsilon > 0$.

3.5.1 Approximate demand bound function

The dynamic program works in phases. During phase φ, $\varphi = 1, 2, \ldots, n$, the DP computes a possible assignment of task τ_φ, using assignments of the first $\varphi - 1$ tasks. In order to obtain a DP table of bounded size, we introduce an approximation of the demand bound function such that the contribution of each task can be derived by using only a constant number of values.

By scaling all parameters, we assume that $d_{\min} = \min_{\tau\in\mathcal{T}} d_\tau = 1$. Let $\epsilon > 0$, and assume without loss of generality that $\epsilon < 1/2$. Let L be the minimum integer that satisfies $1 \leq (1+\epsilon)^{L-1}\epsilon^2$. We define the approximate demand bound function dbf^* as

$$dbf_i^*(\tau, t) := \begin{cases} \left\lfloor \frac{t+p_\tau-d_\tau}{p_\tau} \right\rfloor c_{i,\tau} & \text{if } t < (1+\epsilon)^L d_\tau, \\ \frac{c_{i,\tau}}{p_\tau} t & \text{otherwise.} \end{cases}$$

Given a task assignment $\mathcal{T}$ of the tasks in $\mathcal{T}$ to the machines, we define

$$dbf_{\mathcal{T},i}^*(t) := \sum_{\tau\in\mathcal{T}_i} dbf_i^*(\tau, t) \qquad \forall t > 0.$$

We write $dbf_i^*(t)$ instead of $dbf_{\mathcal{T},i}^*(t)$ in case the assignment $\mathcal{T}$ is clear from the context.

The key observation is that for computing the function $dbf_i^*(\tau, t)$ for some task τ, it suffices to know the utilization of the task τ and the values of the demand bound function $dbf_i(\tau, t)$ for $t \in \left[d_\tau, (1+\epsilon)^L d_\tau\right)$. Exploiting the properties of the functions $dbf_{\mathcal{T},i}(t)$ and $dbf_{\mathcal{T},i}^*(t)$ yields that $dbf_{\mathcal{T},i}^*$ is a $1+\epsilon$ approximation of the "real" demand bound function $dbf_{\mathcal{T},i}$.

Lemma 12. *Given an assignment $\mathcal{T}$ and a constant $\epsilon < 1/2$. Then, for all machines i, (i) if $dbf_{\mathcal{T},i}^*((1+\epsilon)^k) \leq \alpha(1+\epsilon)^k$ for all $k \in \mathbb{N}_{\geq 0}$, then $dbf_{\mathcal{T},i}(t) \leq (1+\epsilon)^2\alpha t$ for all $t \geq 0$; (ii) if $dbf_{\mathcal{T},i}(s) \leq s$ for all $s \geq 0$, then $dbf_{\mathcal{T},i}^*(t) \leq (1+\epsilon)t$ for all $t \geq 0$.*

Proof. For the first claim, we first show that a slightly stronger statement holds for all powers of $1+\epsilon$. We show that if for all $k \in \mathbb{N}_{\geq 0}$ it holds that $dbf^*_{\mathcal{T},i}((1+\epsilon)^k) \leq \alpha(1+\epsilon)^k$, then for all $k \in \mathbb{N}_{\geq 0}$ it holds that

$$dbf_{\mathcal{T},i}((1+\epsilon)^k) \leq (1+\epsilon)\alpha(1+\epsilon)^k. \tag{3.13}$$

Subsequently, we show that for all t it holds that $dbf_{\mathcal{T},i}(t) \leq (1+\epsilon)^2\alpha t$.

Inequality (3.13) trivially holds for all k with $(1+\epsilon)^k \leq d_{\min}$, as then $dbf_i(\tau, (1+\epsilon)^k) = 0$ for all tasks $\tau \in \mathcal{T}$. Assume that (3.13) holds for all $k' \in \mathbb{N}_{\geq 0}$ with $k' < k$, we show that it then also holds for k. Let $t := (1+\epsilon)^k$. Consider some partition of the tasks $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_m\}$. For some time t, let $\mathcal{T}_i^{\text{early}} := \{\tau \in \mathcal{T}_i | (1+\epsilon)^L d_\tau < t\}$ and $\mathcal{T}_i^{\text{late}} := \mathcal{T}_i \backslash \mathcal{T}_i^{\text{early}}$. The tasks in $\mathcal{T}_i^{\text{early}}$ are tasks τ that have their relative deadline so early that time t is larger than $(1+\epsilon)^L d_\tau$. These are the tasks for which $dbf^*_i(\tau, t)$ differs from $dbf_i(\tau, t)$. By definition of $dbf^*_{\mathcal{T},i}$ it follows that

$$dbf_{\mathcal{T},i}(t) \leq \sum_{\tau \in \mathcal{T}_i} dbf^*_i(\tau, t) + \sum_{\tau \in \mathcal{T}_i^{\text{early}}} c_{i,\tau} = dbf^*_{\mathcal{T},i}(t) + \sum_{\tau \in \mathcal{T}_i^{\text{early}}} c_{i,\tau}. \tag{3.14}$$

Further, since for $\tau \in \mathcal{T}_i^{\text{early}}$ it holds that $d_\tau < \frac{t}{(1+\epsilon)^L}$,

$$\begin{aligned}
\sum_{\tau \in \mathcal{T}_i^{\text{early}}} c_{i,\tau} &\leq \sum_{\tau \in \mathcal{T}_i^{\text{early}}} \left\lfloor \frac{\frac{t}{(1+\epsilon)^L} + p_\tau - d_\tau}{p_\tau} \right\rfloor c_{i,\tau} \\
&= \sum_{\tau \in \mathcal{T}_i^{\text{early}}} dbf_i\left(\tau, \frac{t}{(1+\epsilon)^L}\right) \leq dbf_{\mathcal{T},i}\left(\frac{t}{(1+\epsilon)^L}\right) \\
&\overset{(*)}{\leq} \frac{\alpha t}{(1+\epsilon)^{L-1}} \leq \epsilon^2 \alpha t \leq \epsilon \alpha t.
\end{aligned} \tag{3.15}$$

The inequality at $(*)$ is due to the induction hypothesis; we assumed (3.13) to be correct for all $k' < k$ and $\left(\frac{t}{(1+\epsilon)^L}\right) = (1+\epsilon)^{k'}$ for $k' = k - L$. (Note that although this was only shown for $k' \in \mathbb{N}_{\geq 0}$, it holds also if $k' = k - L < 0$, since then $(1+\epsilon)^{k'} \leq 1 = d_{\min}$.) The penultimate inequality follows from $\frac{1}{(1+\epsilon)^{L-1}} < \epsilon^2$. Inequalities (3.14) and (3.15) imply that $dbf_{\mathcal{T},i}(t) \leq dbf^*_{\mathcal{T},i}(t) + \epsilon\alpha t \leq (1+\epsilon)\alpha t$. For all values of t which are *not* powers of $1+\epsilon$, we observe that the function $dbf_{\mathcal{T},i}(t)$ is non-decreasing and thus if (3.13) holds for all $k \in \mathbb{N}_{\geq 0}$, then (i) holds for all t. That is, for values t that are not powers of $1+\epsilon$ we need an additional factor of $1+\epsilon$.

For the second claim, no induction is necessary and we calculate that

$$\begin{aligned}
dbf^*_{\mathcal{T},i}(t) &< \sum_{\tau\in\mathcal{T}_i^{\text{late}}} dbf_i(\tau,t) + \sum_{\tau\in\mathcal{T}_i^{\text{early}}} \frac{t-d_\tau+\frac{t}{(1+\epsilon)^L}}{p_\tau} c_{i,\tau} \\
&\leq \sum_{\tau\in\mathcal{T}_i^{\text{late}}} dbf_i(\tau,t) + \sum_{\tau\in\mathcal{T}_i^{\text{early}}} \left(\left\lfloor \frac{t-d_\tau+p_\tau}{p_\tau} \right\rfloor + \frac{t}{(1+\epsilon)^L p_\tau} \right) c_{i,\tau} \\
&\leq dbf_{\mathcal{T},i}(t) + \sum_{\tau\in\mathcal{T}_i^{\text{early}}} (\epsilon^2 u_{i,\tau}) t \\
&= dbf_{\mathcal{T},i}(t) + \epsilon^2 dbf^*_{\mathcal{T}_i^{\text{early}},i}(t) \\
&\leq dbf_{\mathcal{T},i}(t) + \epsilon^2 dbf^*_{\mathcal{T},i}(t).
\end{aligned}$$

Therefore, $(1-\epsilon^2)dbf^*_{\mathcal{T},i}(t) \leq dbf_{\mathcal{T},i}(t)$, i.e., $dbf^*_{\mathcal{T},i}(t) \leq (1+\epsilon)dbf_{\mathcal{T},i}(t)$, for any $\epsilon < 1/2$. □

Note that, in contrast to other approximations of the demand bound function considered in the literature (e.g., in [2]), in Lemma 12 we do not use an analysis task by task, and we do not bound the ratio $dbf(\tau,t)/dbf^*(\tau,t)$. In fact, the latter can be unbounded: consider for example a task τ with $c_\tau = 1$, $d_\tau = 1$, and $p_\tau = M$ for a very large value M, then $dbf(\tau,t) \geq 1$ for all $t \geq 1$ whereas $dbf^*(\tau,(1+\epsilon)^L) = (1+\epsilon)^L/M$.

Observe that Lemma 12 implies that at the cost of a $(1+\epsilon)^2$ speedup it suffices to check whether the condition $dbf^*_{\mathcal{T},i}(t) \leq t$ is (approximately) satisfied at powers of $1+\epsilon$. Therefore, the DP may characterize each task τ only by its utilization and the constantly many values $dbf^*_i(\tau,(1+\epsilon)^k)$ (namely those values for integers k such that $d_\tau \leq (1+\epsilon)^k < (1+\epsilon)^L d_\tau$), for each machine i.

3.5.2 The dynamic program

For each task τ and each machine i, we introduce a vector $v(i,\tau)$ that stores the (normalized) approximate demand bound function. For all $\ell \in \mathbb{N}_{\geq 0}$ position $v(i,\tau)_\ell$ is defined as

$$v(i,\tau)_\ell := \frac{dbf^*_i(\tau,(1+\epsilon)^\ell)}{(1+\epsilon)^\ell}.$$

Recall that for any vector $\mathbf{a}$ the infinity norm $\|\mathbf{a}\|_\infty = \max_i \{a_i\}$. The following proposition follows by definition.

Proposition 8. *Consider an assignment $\mathcal{T}$. For all machines $i \in M$, it holds that $\left\|\sum_{\tau\in\mathcal{T}_i} v(i,\tau)\right\|_\infty \leq \alpha$ if and only if $dbf^*_{\mathcal{T},i}((1+\epsilon)^\ell) \leq \alpha(1+\epsilon)^\ell$, for all $\ell \in \mathbb{N}_{\geq 0}$.*

We present a dynamic programming algorithm which either (i) asserts that there is no feasible assignment of the tasks to the machines by showing that there is no assignment $\mathcal{T}$ of tasks to machines such that $\left\|\sum_{\tau\in\mathcal{T}_i} v(i,\tau)\right\|_\infty \leq 1+\epsilon$ for each machine i, or (ii) finds an assignment $\mathcal{T}$ such that $\left\|\sum_{\tau\in\mathcal{T}_i} v(i,\tau)\right\|_\infty \leq 1+O(\epsilon)$ for each machine i. In the latter case, Lemma 12 and the above proposition imply an approximation test for the problem

of assigning tasks to a constant number of unrelated machines. The test either concludes that the task system is not feasible (without speedup) or provides an assignment which is feasible in case the machines have a speedup factor of $1 + O(\epsilon)$.

In order to obtain these results, we construct a DP table where each entry represents the subset of the tasks already scheduled, and for each machine information on the load on that machine due to different subsets of the already scheduled tasks. Then, each entry stores either "YES" or "NO", where an entry stores "YES" if there exists an assignment of the tasks corresponding to this entry yielding the loads for all machines specified by the other parameters for this entry. In the remainder of this section we will give the details on the entries of the DP table and how this table is filled. Before doing that, some preprocessing is needed in order to obtain a DP table of bounded size.

Preprocessing

Assume without loss of generality that the tasks $\tau_1, \ldots, \tau_n$ are ordered such that $d_{\tau_\varphi} \leq d_{\tau_{\varphi+1}}$ for each $\varphi \in [n-1]$. We partition the tasks into groups $G_k := \{\tau | (1+\epsilon)^k \leq d_\tau < (1+\epsilon)^{k+1}\}$ for each $k \in \mathbb{N}_{\geq 0}$. The proposed DP works in phases; one phase for each task. The key idea is that when trying to assign task $\tau \in G_k$, the DP needs only a constant number of values from the assignment of the previously considered tasks.

Define $L^{(k)} := \min\{k, L\}$ (such that $k - L^{(k)} \geq 0$). For all tasks having a deadline at most a factor $(1+\epsilon)^L$ smaller than d_{τ_φ}, the DP needs to know how much the vectors of tasks from each group $G_{k'}$ (with $k - L^{(k)} < k' \leq k$) contribute towards dimension ℓ on machine i, for $k \leq \ell \leq k + L$. For the same groups $G_{k'}$ the DP needs to know the summed utilization per machine i over group $G_{k'}$. For the remaining groups $G_{k'}$ with $k' \leq k - L^{(k)}$ (i.e., the tasks that have a deadline at least a factor $(1+\epsilon)^L$ smaller than d_{τ_φ}) for each machine i, only the aggregated utilization over all groups is needed. Summarizing, we need, for all i,

- the sum $\displaystyle\sum_{\tau \in \mathcal{T}_i \cap \left(\bigcup_{k'=0}^{k-L^{(k)}} G_{k'} \right)} u_{i,\tau}$,
- the sum $\displaystyle\sum_{\tau \in \mathcal{T}_i \cap G_{k'}} u_{i,\tau}$, for all k' s.t. $k - L^{(k)} < k' \leq k$, and
- the sum $\displaystyle\sum_{\tau \in \mathcal{T}_i \cap G_{k'}} v(i,\tau)_\ell$, for all ℓ s.t. $k \leq \ell \leq k + L$ and all k' s.t. $\ell - L^{(\ell)} < k' \leq k$.

Ideally, we would like the DP to store all possible combinations of the above quantities that can result from assigning the tasks of previous iterations. Then, the DP could compute the values for the next iteration by taking each combination of values from the last iteration and additionally schedule task τ to one of the machines. Unfortunately, the number of possible combinations of the above values is not polynomially bounded, as already the input values (which then imply the utilization, etc.) might be in an exponential range. In order to bound them, we round entries of the vectors $v(i,\tau)$. The DP then performs the described procedure with the rounded vectors. This will result in a polynomial-time procedure.

Consider a group G_k, for $k \in \mathbb{N}_{\geq 0}$. For all i and $\tau \in G_k$, define $v'(i,\tau)_\ell := \frac{\epsilon}{n} \left\lfloor \frac{n}{\epsilon} v(i,\tau)_\ell \right\rfloor$ for each $\ell \leq k + L$, and $v'(i,\tau)_\ell := u'_{i,\tau} := \frac{\epsilon}{n} \left\lfloor \frac{n}{\epsilon} u_{i,\tau} \right\rfloor$ for each $\ell > k + L$. The following lemma bounds the rounding error.

Lemma 13. *Let i be a machine and $\mathcal{T}_i$ be a set of tasks. For all $\ell \in \mathbb{N}_{\geq 0}$, it holds that $\sum_{\tau \in \mathcal{T}_i} v'(i,\tau)_\ell \leq \sum_{\tau \in \mathcal{T}_i} v(i,\tau)_\ell \leq \sum_{\tau \in \mathcal{T}_i} v'(i,\tau)_\ell + \epsilon$.*

Proof. Consider a task $\tau \in \mathcal{T}_i$. Define $k(\tau)$ such that $\tau \in G_{k(\tau)}$. We show for each $\tau \in \mathcal{T}_i$ and the corresponding $k(\tau)$ that $v'(i,\tau)_\ell \leq v(i,\tau)_\ell \leq v'(i,\tau)_\ell + \epsilon/n$. The statement trivially follows. The case where $\ell \leq k(\tau) + L$ follows trivially from the relation $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$ which holds for any x. Therefore, consider the case $\ell > k(\tau) + L$. Since $\tau \in G_{k(\tau)}$, it follows that $d_\tau < (1+\epsilon)^{k(\tau)+1}$. Hence, $dbf_i^*(\tau, (1+\epsilon)^\ell) = \frac{c_{i,\tau}}{p_\tau}(1+\epsilon)^\ell = u_{i,\tau}(1+\epsilon)^\ell$ which yields $v(i,\tau)_\ell = u_{i,\tau}$. The result for the case $\ell > k(\tau) + L$ now also follows from $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$. □

Note that each rounded vector $v'(i,\tau)$ can be described with only constantly many pieces of information. When working with the rounded vectors, for the quantities mentioned above, there are only a polynomial number of combinations (assuming that m is a constant). In particular, the dynamic-programming table will be of polynomial size. In the next subsection we describe what the entries of the DP will be.

Entries of the DP table

The dynamic-programming table consists of entries of the form $(\varphi, \boldsymbol{\mu}, \boldsymbol{\nu}, \boldsymbol{\xi})$ where

- $\varphi \in \{0, \ldots, n\}$ denotes the phase of the DP. In phase φ, task τ_φ is being assigned to a machine. Let k be an integer such that $\tau_\varphi \in G_k$;
- in phase φ, for each machine i, the value $\mu_i^{(\varphi)}$ is of the form $q\frac{\epsilon}{n}$ for some integer q, denoting the rounded aggregated utilization for machine i due to the tasks having a deadline at least a factor of $(1+\epsilon)^L$ smaller with respect to the deadline of task τ_φ;
- in phase φ, for each machine i and each k' with $k - L^{(k)} < k' \leq k$, the value $\nu_{i,k'}^{(\varphi)}$ is of the form $q\frac{\epsilon}{n}$ for some integer q, denoting the rounded utilization of tasks in $G_{k'} \cap \mathcal{T}_i$;
- in phase φ, for each triple $(i,k',k'') \in C_\varphi$ with $C_\varphi = \{(i,k',k'') : 1 \leq i \leq m;\ k \leq k'' < k + L \text{ and } k'' - L^{(k'')} < k' \leq k\}$, the value $\xi_{i,k',k''}^{(\varphi)}$ is of the form $q\frac{\epsilon}{n}$ for some integer q, denoting the quantity $\sum_{\tau \in \mathcal{T}_i \cap G_{k'}} v'(i,\tau)_{k''}$, expressing how much the vectors of the tasks in $G_{k'}$ on machine i contribute towards dimension k''.

We require the following set of conditions to be satisfied for a DP cell $(\varphi, \boldsymbol{\mu}, \boldsymbol{\nu}, \boldsymbol{\xi})$ to exist; for each machine $i \in M$ and all $k'' \in \{k, \ldots, k+L\}$

$$\mu_i + \sum_{k'=k-L^{(k)}+1}^{k''-L^{(k'')}} \nu_{i,k'} + \sum_{k'=k''-L^{(k'')}+1}^{k} \xi_{i,k',k''} \leq 1 + \epsilon. \tag{3.16}$$

This condition implies that, for all parameters, μ_i, $\nu_{i,k'}$, $\xi_{i,k',k''} \leq 1 + \epsilon$.

Proposition 9. *The number of DP cells is bounded by* $n((1+\epsilon)n/\epsilon)^{2mL^2}$.

Proof. The values μ_i, $\nu_{i,k'}$ and $\xi_{i,k',k''}$ are all stored with accuracy $\frac{\epsilon}{n}$ and hence each of those can take $(1+\epsilon)n/\epsilon$ many different values. Further, i can take m different values whereas k' and k'' can take L different values if the DP is in a certain phase φ. Further, there are at most n phases for the DP. It follows that the number of cells of the DP table is no more than

$$n \cdot \left(\frac{(1+\epsilon)n}{\epsilon}\right)^{m} \cdot \left(\frac{(1+\epsilon)n}{\epsilon}\right)^{mL} \cdot \left(\frac{(1+\epsilon)n}{\epsilon}\right)^{mL^2} \leq n\left(\frac{(1+\epsilon)n}{\epsilon}\right)^{2mL^2} \qquad \square$$

Each entry $(\varphi, \boldsymbol{\mu}, \boldsymbol{\nu}, \boldsymbol{\xi})$ of the DP table either stores "YES" or "NO" depending on whether or not there is an assignment of the tasks $\tau_1, \ldots, \tau_\varphi$ to the machines which yields the quantities given by the vectors $\boldsymbol{\mu}$,$\boldsymbol{\nu}$ and $\boldsymbol{\xi}$.

Filling the DP table

We proceed by describing how to fill the DP table. First, initialize the table by assigning a 'YES' entry to $(0, \mathbf{0}, \mathbf{0}, \mathbf{0})$ and a 'NO' entry to any other entry with $\varphi = 0$. Assume that for some φ, all entries of the form $(\varphi - 1, \boldsymbol{\mu}^{(\varphi-1)}, \boldsymbol{\nu}^{(\varphi-1)}, \boldsymbol{\xi}^{(\varphi-1)})$ have been computed. The DP table is then iteratively extended to phase φ. Phase φ considers each combination of assigning task τ_φ to some machine i and a DP cell $(\varphi - 1, \boldsymbol{\mu}^{(\varphi-1)}, \boldsymbol{\nu}^{(\varphi-1)}, \boldsymbol{\xi}^{(\varphi-1)})$ with a 'YES' entry. Intuitively, the DP computes which values for $\boldsymbol{\mu}^{(\varphi)}$, $\boldsymbol{\nu}^{(\varphi)}$, and $\boldsymbol{\xi}^{(\varphi)}$ are obtained if it takes the task assignment encoded in the DP cell $(\varphi - 1, \boldsymbol{\mu}^{(\varphi-1)}, \boldsymbol{\nu}^{(\varphi-1)}, \boldsymbol{\xi}^{(\varphi-1)})$ and additionally schedules task τ_φ to machine i.

Let tasks $\tau_{\varphi-1}$ and τ_φ be in group G_h and G_k, respectively. Almost all entries of the vectors remain equal when going to phase φ and hence we only list the values which differ with respect to the entries from phase $\varphi - 1$. If $h = k$ and task τ_φ is assigned to machine i, then $\nu_{i,k}^{(\varphi)} = \nu_{i,k}^{(\varphi-1)} + u'_{i,\tau_\varphi}$, and $\xi_{i,k,k''}^{(\varphi)} = \xi_{i,k,k''}^{(\varphi-1)} + v'(i,\tau_\varphi)_{k''}$ for all $k'' \in \{k, \ldots, k+L\}$.

If $h \neq k$, w.l.o.g. we assume that $h = k - 1$ (e.g., by creating dummy tasks of zero processing requirement to fill in-between groups that otherwise would be empty and thus, non-existent). Then, $\mu_g^{(\varphi)} = \mu_g^{(\varphi-1)} + \nu_{g,k-L^{(k)}}^{(\varphi-1)}$ for all machines $g \in M$; as we have moved up one group, tasks from group $G_{k-L^{(k)}}$ now have their deadline at least a factor $(1+\epsilon)^L$ away from d_{τ_φ} and their utilizations are not stored separately anymore, but in a aggregated way. For the tasks in groups $G_{k'}$ such that $k - L^{(k)} < k' < k$ and all machines $g \in M$ we still store the utilizations per group and thus $\nu_{g,k'}^{(\varphi)} = \nu_{g,k'}^{(\varphi-1)}$. For group G_k we also store the utilization for the machine i that it was assigned to, so $\nu_{i,k}^{(\varphi)} = u'_{i,\tau_\varphi}$ and $\nu_{g,k}^{(\varphi)} = 0$ for all machines $g \neq i$. Since τ_φ is assigned to machine i, $\xi_{i,k,k''}^{(\varphi)} = v'(i,\tau_\varphi)_{k''}$ for all $k'' : k \leq k'' \leq k+L$; and for all machines $g \neq i$, $\xi_{g,k,k''}^{(\varphi)} = 0$ for all $k'' : k \leq k'' \leq k+L$. Finally, $\xi_{g,k',k''}^{(\varphi)} = \xi_{g,k',k''}^{(\varphi-1)}$ for all machines $g \in M$, all $k'' : k \leq k'' \leq k+L$ and all $k' : k'' - L^{(k'')} < k' < k$.

Hereafter, the DP checks whether the computed values $\boldsymbol{\mu}^{(\varphi)}$, $\boldsymbol{\nu}^{(\varphi)}$ and $\boldsymbol{\xi}^{(\varphi)}$ satisfy the condition given in (3.16). If this is the case, then the corresponding DP cell $(\varphi, \boldsymbol{\mu}^{(\varphi)}, \boldsymbol{\nu}^{(\varphi)}, \boldsymbol{\xi}^{(\varphi)})$ is filled with a 'YES' entry and we say that this DP cell *extends* the DP cell $(\varphi - 1, \boldsymbol{\mu}^{(\varphi-1)}, \boldsymbol{\nu}^{(\varphi-1)}, \boldsymbol{\xi}^{(\varphi-1)})$. In case there does not exist a DP cell

$(\varphi - 1, \boldsymbol{\mu}^{(\varphi-1)}, \boldsymbol{\nu}^{(\varphi-1)}, \boldsymbol{\xi}^{(\varphi-1)})$ which can be extended to the DP cell $(\varphi, \boldsymbol{\mu}^{(\varphi)}, \boldsymbol{\nu}^{(\varphi)}, \boldsymbol{\xi}^{(\varphi)})$, the latter DP cell is filled with a 'NO' entry.

The DP table is filled inductively, phase by phase, until each cell in the DP table is filled. In the next two lemmas we show for any machine i the equivalence between the inequality $\left\|\sum_{\tau\in\mathcal{T}_i} v'(i,\tau)\right\|_\infty \leq 1+\epsilon$ on the one hand, and condition (3.16) on the other hand.

Lemma 14. *For phase φ, if there exists a DP cell of the form $(\varphi, \boldsymbol{\mu}^{(\varphi)}, \boldsymbol{\nu}^{(\varphi)}, \boldsymbol{\xi}^{(\varphi)})$ with a 'YES' entry, then there exists a task assignment $\mathcal{T}$ of the first φ tasks to the machines, such that for each $i \in M$ it holds that $\left\|\sum_{\tau\in\mathcal{T}_i} v'(i,\tau)\right\|_\infty \leq 1+\epsilon$.*

Proof. Consider some phase φ and assume that the statement is correct for all phases $\varphi' < \varphi$. Let k be such that $\tau_\varphi \in G_k$. As no first job of any task in G_k has its deadline before $(1+\epsilon)^k$, it follows by induction that for all ℓ such that $0 \leq \ell < k$ it holds that $\sum_{\tau\in\mathcal{T}_i} v'(i,\tau)_\ell \leq 1+\epsilon$. Next, we show the statement for phase φ and the corresponding dimension k. Consider DP cell of the form $(\varphi, \boldsymbol{\mu}^{(\varphi)}, \boldsymbol{\nu}^{(\varphi)}, \boldsymbol{\xi}^{(\varphi)})$ with a 'YES' entry. We denote by $(3.16)^{(\varphi)}$ the corresponding inequality given in (3.16) at phase φ of the DP, that is, when task τ_φ is being assigned to a machine. For all machines i it follows that,

$$\begin{aligned}\sum_{\tau\in\mathcal{T}_i} v'(i,\tau)_k &= \sum_{k'=0}^{k-L^{(k)}} \sum_{\tau\in\mathcal{T}_i\cap G_{k'}} u'_{i,\tau} + \sum_{k'=k-L^{(k)}+1}^{k} \sum_{\tau\in\mathcal{T}_i\cap G_{k'}} v'(i,\tau)_k \\ &= \mu_i^{(\varphi)} + \sum_{k'=k-L^{(k)}+1}^{k} \xi_{i,k',k}^{(\varphi)} \overset{(3.16)^{(\varphi)}}{\leq} 1+\epsilon.\end{aligned}$$

The last inequality follows by setting the parameter k'' of $(3.16)^{(\varphi)}$ equal to k. Next, consider the dimensions $\ell : k < \ell \leq k+L$. Note that $d_{\tau_\varphi} < (1+\epsilon)^{k+1}$. Consequently, every task in $\cup_{k'=0}^{k} G_{k'}$ has its first deadline before $(1+\epsilon)^{k+1}$, that is, while being in phase φ of the DP where $\tau_\varphi \in G_k$ it follows that $G_{k'} = \emptyset$ for all $k' > k$. This insight is used in the first equality below.

$$\begin{aligned}\sum_{\tau\in\mathcal{T}_i} v'(i,\tau)_\ell &= \sum_{k'=0}^{\ell-L^{(\ell)}} \sum_{\tau\in\mathcal{T}_i\cap G_{k'}} u'_{i,\tau} + \sum_{k'=\ell-L^{(\ell)}+1}^{k} \sum_{\tau\in\mathcal{T}_i\cap G_{k'}} v'(i,\tau)_\ell \\ &= \left(\mu_i^{(\varphi)} + \sum_{k'=k-L^{(k)}+1}^{\ell-L^{(\ell)}} \nu_{i,k'}^{(\varphi)}\right) + \sum_{k'=\ell-L^{(\ell)}+1}^{k} \xi_{i,k',\ell}^{(\varphi)} \overset{(3.16)^{(\varphi)}}{\leq} 1+\epsilon.\end{aligned}$$

The last inequality follows by setting the parameter k'' of $(3.16)^{(\varphi)}$ equal to ℓ. Finally, the analysis for any dimension $\ell > k+L$ is equal to that of the dimension $\ell = k+L$ as the contribution of each task to any dimension $\ell \geq k+L$ will be approximated by its utilization. Thus, $\left\|\sum_{\tau\in\mathcal{T}_i} v'(i,\tau)\right\|_\infty \leq 1+\epsilon$ and the statement follows. □

Lemma 15. *For phase φ, there exists a DP cell of the form $(\varphi, \boldsymbol{\mu}^{(\varphi)}, \boldsymbol{\nu}^{(\varphi)}, \boldsymbol{\xi}^{(\varphi)})$ with a 'YES' entry, if there exists a task assignment $\mathcal{T}$ of the first φ tasks to the machines, such that for each $i \in M$ it holds that $\left\|\sum_{\tau\in\mathcal{T}_i} v'(i,\tau)\right\|_\infty \leq 1+\epsilon$.*

Proof. Consider the assignment $\mathcal{T}^{(\varphi)}$ where the first φ tasks are assigned to the machines such that $\left\|\sum_{\tau\in\mathcal{T}_i^{(\varphi)}} v'(i,\tau)\right\|_\infty \leq 1+\epsilon$ for all machines $i \in M$. Let $\tau_\varphi \in G_k$. Define the following values for each machine i:

$$\begin{aligned}
\mu_i^{(\varphi)} &:= \sum_{k'=0}^{k-L^{(k)}} \sum_{\tau\in\mathcal{T}_i^{(\varphi)}\cap G_{k'}} u'_{i,\tau};\\
\nu_{i,k'}^{(\varphi)} &:= \sum_{\tau\in\mathcal{T}_i^{(\varphi)}\cap G_{k'}} u'_{i,\tau} \qquad \text{for } k' \in \{k-L^{(k)},\ldots,k\};\\
\xi_{i,k',k''}^{(\varphi)} &:= \sum_{\tau\in\mathcal{T}_i^{(\varphi)}\cap G_{k'}} v'(i,\tau)_{k''} \qquad \text{for } k'' \in \{k,\ldots,k+L\},\ \text{for } k' \in \{k''-L^{(k'')},\ldots,k\}.
\end{aligned}$$

First we need to show that the DP cell $(\varphi, \boldsymbol{\mu}^{(\varphi)}, \boldsymbol{\nu}^{(\varphi)}, \boldsymbol{\xi}^{(\varphi)})$ exists, that is, we need to check whether inequality $(3.16)^{(\varphi)}$ holds. From the lemma statement, it is known that $\sum_{\tau\in\mathcal{T}_i} v'(i,\tau)_\ell \leq 1+\epsilon$, for all ℓ. Thus, in particular, this inequality also holds for all dimensions $k'' \in \{k,\ldots,k+L\}$. Therefore, for each machine i and each dimension $k'' \in \{k,\ldots,k+L\}$ it holds that

$$\begin{aligned}
\mu_i^{(\varphi)} + \sum_{k'=k-L^{(k)}+1}^{k''-L^{(k'')}} \nu_{i,k'}^{(\varphi)} + \sum_{k'=k''-L^{(k'')}+1}^{k} \xi_{i,k',k''}^{(\varphi)}&\\
= \sum_{k'=0}^{k''-L^{(k'')}} \sum_{\tau\in G_{k'}} v'(i,\tau)_{k''} + \sum_{k'=k''-L^{(k'')}+1}^{k} \xi_{i,k',k''}^{(\varphi)}&\\
= \sum_{k'=0}^{k''-L^{(k'')}} \sum_{\tau\in G_{k'}} v'(i,\tau)_{k''} + \sum_{k'=k''-L^{(k'')}+1}^{k} \sum_{\tau\in G_{k'}} v'(i,\tau)_{k''}&\\
= \sum_{\tau\in\mathcal{T}_i^{(\varphi)}} v'(i,\tau)_{k''} \leq 1+\epsilon.&
\end{aligned}$$

Consequently, the DP cell $(\varphi, \boldsymbol{\mu}^{(\varphi)}, \boldsymbol{\nu}^{(\varphi)}, \boldsymbol{\xi}^{(\varphi)})$ exists. The DP cell $(\varphi, \boldsymbol{\mu}^{(\varphi)}, \boldsymbol{\nu}^{(\varphi)}, \boldsymbol{\xi}^{(\varphi)})$ trivially extends the DP cell $(\varphi-1, \boldsymbol{\mu}^{(\varphi-1)}, \boldsymbol{\nu}^{(\varphi-1)}, \boldsymbol{\xi}^{(\varphi-1)})$ by assigning task τ_φ to machine i if $\tau_\varphi \in \mathcal{T}_i^{(\varphi)}$. By induction it follows that the DP cell $(\varphi, \boldsymbol{\mu}^{(\varphi)}, \boldsymbol{\nu}^{(\varphi)}, \boldsymbol{\xi}^{(\varphi)})$ contains a 'YES' entry, for all $\varphi \in \{0,1,\ldots,n\}$. □

A PTAS feasibility test

Combining Lemmas 12, 13, 14 and 15, and Proposition 8 yields a $(1+7\epsilon)$-approximation test, for any ϵ and a constant number of machines. The claim on the running time follows from Proposition 9. If for a given $\epsilon > 0$ we run the above procedure with $\epsilon' := \epsilon/7$, rather than with ϵ, we obtain a $(1+\epsilon)$-approximation test.

Theorem 7. *For a constant number of machines and for any $\epsilon > 0$ there exists a $(1+\epsilon)$-approximation test, that runs in time polynomial in the number of tasks.*

Proof. We show that for a constant number of unrelated machines, the algorithm given above either concludes that the task system is infeasible, or returns an assignment of tasks to machines that is feasible with a speedup factor of $1 + 7\epsilon$. The running time is polynomial in n, given that m and ϵ are constants. First, we redefine $\epsilon = \min\{\epsilon, 1/2\}$.

Suppose that there is a DP cell of the form $(n, \boldsymbol{\mu}, \boldsymbol{\nu}, \boldsymbol{\xi})$ containing a 'YES' entry. Due to Lemma 14 there is an assignment $\mathcal{T}$ of all tasks to the machines such that $\left\|\sum_{\tau \in \mathcal{T}_i} v'(i,\tau)\right\|_\infty \leq 1 + \epsilon$, for each machine $i \in M$. By Lemma 13 this implies that $\left\|\sum_{\tau \in \mathcal{T}_i} v(i,\tau)\right\|_\infty \leq 1 + 2\epsilon$. Due to Proposition 8 this implies that $dbf^*_{\mathcal{T},i}((1+\epsilon)^k) \leq (1+2\epsilon)(1+\epsilon)^k$ for each $k \in [n]$. Finally, Lemma 12 implies that the computed task assignment is feasible if the machines run with speed $(1+\epsilon)^2(1+2\epsilon) \leq 1+7\epsilon$ (as $\epsilon \leq 1/2$).

On the other hand, if all DP cells of the form $(n, \boldsymbol{\mu}, \boldsymbol{\nu}, \boldsymbol{\xi})$ have a 'NO' entry, then, by Lemma 15, for any task assignment $\mathcal{T}$ there must be a machine i with $\left\|\sum_{\tau \in \mathcal{T}_i} v'(i,\tau)\right\|_\infty > 1 + \epsilon$. By Lemma 13 this yields then that also $\left\|\sum_{\tau \in \mathcal{T}_i} v(i,\tau)\right\|_\infty > 1 + \epsilon$. Proposition 8 yields that for this machine i there exists a time instant t which is a power of $(1+\epsilon)$ such that $dbf^*_{\mathcal{T},i}(t) > (1+\epsilon)t$. Finally, (the negation of) Lemma 12 yields that there is a time instant s for which $dbf_{\mathcal{T},i}(s) > s$. Since, for any partition $\mathcal{T}$, there exist a machine i and a time instant s such that $\mathcal{T}$ violates the feasibility condition, it follows that the task system $\mathcal{T}$ is infeasible if the machines run at unit speed.

The claim that the running time is polynomial in n for given constant m and ϵ follows from Proposition 9 and the fact that each entry of the table can be decided upon in polynomial time. Herewith, note that $L = O(\log_{(1+\epsilon)}(1/\epsilon^2))$, which is constant if ϵ is constant. □

3.6 Epilogue

In this chapter we present the first results for assigning sporadic tasks with arbitrary deadlines to unrelated parallel processors. Through the development of a new LP rounding procedure and approximations of the *demand bound function* we found a $8 + 2\sqrt{6} \approx 12.9$-approximate feasibility test for an arbitrary number of machines. We hope that future research might bring the approximation ratio down and close the gap between 12.9 and our lower bound of 2. One possible tool might be configuration LPs which are often stronger than assignment LPs like the one that we use here (see, e.g., [5, 10, 69]). Another interesting direction would be to obtain better approximations for the case that the processors are unrelated but there are only a few types of processors (e.g., CPUs and GPUs), as done in, e.g., [61, 62, 72].

Our rounding procedure is very general and can not only be applied to the problem of assigning tasks to machines but for any assignment problem which allows for a sparse linear program. It would be interesting to see other new applications for it. Additionally, it would be interesting if a better rounding procedure can be given for large values of γ.

For a constant number of machines we give a polynomial-time approximation scheme. While for scheduling jobs on unrelated machines, a PTAS is relatively easy to obtain once one has the tools from Lenstra, Shmoys and Tardos [53] at hand, for assigning sporadic tasks, much more sophisticated machinery was required.

Chapter 4

Real-time tasks on identical machines

All results in this chapter appear in [9].

4.1 Introduction

We consider a setting with multiple machines and a task set that we want to partition over the machines. As opposed to the previous chapter, the machines are now assumed to be *identical*. We denote by m the number of machines. The problem of partitioning a sporadic task set over m machines is a co-NP-hard problem [37] and hence we need to find a good approximation algorithm. Recall that for a given parameter $\alpha \geq 1$, an α-approximate feasibility test either returns a partition of the tasks into sets $\{\mathcal{T}_i\}_{1 \leq i \leq m}$ such that each set $\mathcal{T}_i$ can be feasibly scheduled on a machine that runs at speed α, or returns "infeasible" if no feasible partition of the tasks exists that can be scheduled on m machines running at unit speed.

Recall that we call a family of feasibility tests a polynomial-time approximation scheme (PTAS), if for any arbitrarily small constant $\epsilon > 0$, there exists a $(1+\epsilon)$-approximate feasibility test in this family with running time polynomial in n and m (with n being the number of tasks and m the number of machines). Note that the running time dependence on ϵ can be any arbitrary function. If the running time dependence on $1/\epsilon$ is also polynomial, we call the test a fully polynomial-time approximation scheme (FPTAS).

4.1.1 Related work

In the single-processor case, an FPTAS feasibility test is known [26]. In particular, in this test, one only checks the feasibility of the EDF schedule for the job sequence at about $(1/\epsilon)\log(d_{\max}/d_{\min})$ time steps, where $d_{\max}$ and $d_{\min}$ are the largest and the smallest relative deadline, respectively.

For partitioned scheduling on multiple machines, Chen and Chakraborty [30] gave a PTAS, generalizing a previous result of [17], for the case that the maximum to minimum deadline ratio is a constant. The idea of [30] is to view the problem as a *vector scheduling* problem in (roughly) $D = (1/\epsilon)\log(d_{\max}/d_{\min})$ dimensions.

Definition 3 (Vector Scheduling)**.** *In the* Vector Scheduling *problem we are given a set V of n rational D-dimensional vectors $v_1, \ldots, v_n$ from $[0,1]^D$ and a positive integer m. The goal is to determine whether there is a partition of V into m sets $V_1, \ldots, V_m$ such that for each set V_i, the sum of vectors in that set does not exceed 1 in any coordinate. That is, such that $\max_{1 \le i \le m} \left\| \sum_{v \in V_i} v \right\|_\infty \le 1$.*

This problem is a D-dimensional generalization of the makespan minimization problem, where each job is a D-dimensional vector and the machines are D-dimensional objects as well. Chekuri and Khanna [27] give the following result for the vector scheduling problem.

Theorem 8 ([27])**.** *Given any fixed $\epsilon > 0$, for the vector scheduling problem there exists a $(1+\epsilon)$-approximation algorithm, i.e., an algorithm that finds a partition $V_1, \ldots, V_m$ such that $\max_{1 \le i \le m} \left\| \sum_{v \in V_i} v \right\|_\infty \le 1 + \epsilon$, if a solution with $\max_{1 \le i \le m} \left\| \sum_{v \in V_i} v \right\|_\infty \le 1$ exists. This algorithms runs in time $(nD/\epsilon)^{O(\psi)}$, where $\psi = O\left(\left(\frac{\ln(D/\epsilon)}{\epsilon}\right)^D\right)$.*

Chen and Chakraborty [30] view each task as a D-dimensional vector, and the tasks can be feasibly scheduled on a machine, if the corresponding vectors can be feasibly packed in a unit D-dimensional bin. This connection essentially follows from the property for the single-processor test mentioned above [26]. Then, the PTAS for vector scheduling [27] is used in a black-box manner to obtain a $(1+\epsilon)$-approximate feasibility test that runs in time roughly[1] $n^{O(\exp((\frac{1}{\epsilon})\log(d_{\max}/d_{\min})))}$. Note that this running time is doubly exponential in $\log(d_{\max}/d_{\min})$, and while this is polynomial time for constant ratios $d_{\max}/d_{\min}$, it is super-polynomial if $d_{\max}/d_{\min}$ is super-constant.

4.1.2 Our results

Let us denote from now on the maximum to minimum deadline ratio $d_{\max}/d_{\min}$ by λ. We provide a $(1+\epsilon)$-approximate feasibility test which substantially improves upon the result of Chen and Chakraborty [30]. The running time of our feasibility test is $O\left(m^{O(f(\epsilon)\log\lambda)}\right)$, where $f(\epsilon) := \exp(O(\frac{1}{\epsilon}\log(\frac{1}{\epsilon})))$ is a function depending solely on ϵ.

Note that the running time of our algorithm only has a singly exponential dependence on $\log\lambda$, and hence it gives an exponential improvement over the result of Chen and Chakraborty [30]. Thus, our algorithm can run over a substantially wider range of input instances, beyond just the ones with $\lambda = O(1)$. For example, even if λ is polynomially large in n (where n denotes the number of tasks), our algorithm runs in time $n^{O(\log n)} = 2^{O(\log^2 n)}$ and hence yields a quasi-polynomial-time approximation scheme, as opposed to exponential time for the algorithm in [30].

As in Chen and Chakraborty [30], our result is also based on reducing the feasibility problem to vector scheduling in roughly $D = (1/\epsilon)\log\lambda$ dimensions. However, we crucially exploit the special structure of the vectors that arise in this transformation and give a faster vector scheduling algorithm for such instances. In fact, it was shown [11] that exploiting this structure is necessary to obtain any major improvement. In particular,

[1] For clarity, we suppress some dependence on terms involving $\log\log(d_{\max}/d_{\min})$.

in [11] it is shown that any PTAS for a general D-dimensional vector scheduling problem must incur a running time of $\exp((1/\epsilon)^{\Omega(D)})$ (under suitable complexity assumptions), and hence the running time in [30] is essentially the best one can hope for if one uses vector scheduling as a black box.

The idea of our algorithm is as follows. A vector in the vector scheduling problem represents for a task τ the (normalized) demand bound function in D different time points. Then, a partitioning $V_1, \ldots, V_m$ of the vectors implies a partitioning $\mathcal{T}_1, \ldots, \mathcal{T}_m$ of the task set $\mathcal{T}$. If the sum of all vectors in V_i is at most $1 + \epsilon$ for each coordinate, this means that the corresponding tasks in $\mathcal{T}_i$ can be scheduled on machine i such that $dbf_{\mathcal{T}}(t) \leq (1 + \epsilon)t$ for all time points t that are represented in one of the coordinates. There are, however, too many time points to check, so in the end, this will only be approximately true.

Many steps are needed to obtain the algorithm as described here. The vectors that we start with have $(1/\epsilon) \log \lambda$ coordinates, but we will reduce the number of coordinates that are relevant to essentially $1/\epsilon$ many. In particular, only $1/\epsilon$ consecutive coordinates of a vector will have "arbitrary" values, and all subsequent coordinates have an identical value.

To exploit the structure of the vectors created, we design a sliding-window based algorithm for vector scheduling, where we carefully build a schedule by considering the coordinates in left to right order, and only keeping track of the relevant short-range information in a dynamic program. The main technical difficulty is to combine the sliding-window approach with the exhaustive enumeration techniques of [27] for vector scheduling. In particular, to ensure that the sliding window does not build up too much error as it moves over the various coordinates, we keep track of different coordinates for a task with different accuracy. Moreover, to keep the running time low, we need more refined enumeration techniques to handle and combine small and large vectors.

4.2 Preliminaries

The input consists of a sporadic task system $\mathcal{T}$ consisting of tasks $\tau_1, \ldots, \tau_n$ and a set of m identical processors. Let us denote by $[k]$ all integers from 1 until k, for some integer k. That is, $[k] := \{1, \ldots, k\}$.

Recall that in the partitioned scheduling paradigm, we want to find a partition $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$ of $\mathcal{T}$ such that all jobs generated by the tasks in $\mathcal{T}_i$ can be feasibly scheduled on machine i, for all $i \in [m]$. Recall Proposition 2 from Chapter 1 which states that $dbf_{\mathcal{T}_i}(t) \leq t$, for all $t \geq 0$, is a necessary and sufficient condition for task set $\mathcal{T}_i$ to be feasible upon machine i.

As mentioned above, our goal is to develop a $(1 + \epsilon)$-approximate feasibility test for any $\epsilon > 0$. As we shall see soon, if we only care about $1 + \epsilon$ feasibility, it suffices to check the demand bound function at only $\log_{(1+\epsilon)}(d_{\max}/(\epsilon^2 d_{\min})) \approx O(\log(d_{\max}/d_{\min})/\epsilon)$ time points. This allows to transform the feasibility problem into the so-called vector scheduling problem, as defined in Section 4.1.1.

In the following section, we show how $1 + \epsilon$ feasibility reduces to vector scheduling with dimension $O((1/\epsilon) \log(d_{\max}/d_{\min}))$. While similar results have been used before (e.g., [30, 58]), we will repeat the proof here, as we explicitly need the structure of the vectors in

the resulting vector scheduling instance, which our algorithm will crucially exploit later.

4.3 Task systems and vector scheduling

Observe that over the long run, a task τ uses c_τ units of time every p_τ units of time, but the relative deadlines, that may be different from the periods, complicate the demand bound function. The demand bound function has sharp jumps at the (absolute) deadlines $d_\tau, p_\tau + d_\tau, 2p_\tau + d_\tau, \ldots$, but the effects of these jumps become milder as time progresses. A machine that is $1+\epsilon$ times faster gives $\epsilon\, t$ units of extra processing time up to time t, which allows ignoring these sharp jumps after a certain point in time and instead it is sufficient to use the utilization $u_\tau = \frac{c_\tau}{p_\tau}$.

The next lemma shows that it is approximately sufficient to check the demand bound function only at time points which are a factor $1+\epsilon$ apart.

Lemma 16. *For any task τ, if $dbf_\tau((1+\epsilon)^k d_{\min}) \le (1+\epsilon)^k d_{\min}\alpha$ for all $k \in \mathbb{N}_{\ge 0}$, then $dbf_\tau(t) \le (1+\epsilon)\alpha t$ for all $t \ge 0$.*

Proof. For any t, define integer k_t such that $(1+\epsilon)^{k_t-1} d_{\min} < t \le (1+\epsilon)^{k_t} d_{\min}$. Then

$$dbf_\tau(t) \le dbf_\tau((1+\epsilon)^{k_t} d_{\min}) \le (1+\epsilon)^{k_t} d_{\min}\alpha < (1+\epsilon)\alpha t,$$

where the first inequality follows from the demand bound function being non-decreasing. □

We will use the same approximate demand bound function $dbf^*_\tau(t)$ used in Chapter 3 (see also Marchetti-Spaccamela et al. [58]). Let L be the smallest integer such that $1 \le (1+\epsilon)^{L-1}\epsilon^2$. Note that $L \le 2 + \log_{(1+\epsilon)}(1/\epsilon^2)$. Let

$$dbf^*_\tau(t) = \begin{cases} \left\lfloor \frac{t+p_\tau-d_\tau}{p_\tau} \right\rfloor c_\tau & \text{if } t < (1+\epsilon)^L d_\tau, \\ u_\tau\, t & \text{otherwise.} \end{cases} \tag{4.1}$$

Note that dbf^* differs from dbf only when $t \ge d_\tau(1+\epsilon)/\epsilon^2$, and is proportional to the utilization of τ in that case. The following lemma shows that it is a good approximation to dbf.

Lemma 17 (see Chapter 3)**.** *For every task τ and every time $t \ge 0$,*

$$\frac{1}{(1+\epsilon)} dbf_\tau(t) \le dbf^*_\tau(t) \le (1+\epsilon) dbf_\tau(t). \tag{4.2}$$

Another obvious property of dbf and dbf^* is the following, which allows us to start our feasibility analysis at the first deadline only.

Observation 1. *For all tasks τ, for all $t < d_\tau$, we have that*

$$dbf_\tau(t) = dbf^*_\tau(t) = \max\{0, \lfloor (t + p_\tau - d_\tau)/p_\tau \rfloor\} = 0.$$

In particular, $dbf_\tau(t) = dbf^*_\tau(t) = 0$ for all $t < d_{\min}$ and all tasks $\tau \in \mathcal{T}$.

Using Lemma 16, Lemma 17 and Observation 1, we can encode our approximate demand bound function dbf^*_τ into a vector w^τ. More precisely, we will use a *normalized* demand bound function which is $dbf^*_\tau(t)/t$. Define $t_k := (1+\epsilon)^k d_{\min}$, $t_{\max} := (1+\epsilon)^L d_{\max}$, and let $K := \lceil \log_{(1+\epsilon)}(t_{\max}/d_{\min}) \rceil$.

For each task τ we define the vector w^τ, with coordinates w^τ_k as follows:

$$w^\tau_k := \begin{cases} \frac{dbf^*_\tau(t_{k-1})}{t_{k-1}} & \text{if } k = 1, \ldots, K-1, \\ u_\tau & \text{if } k = K. \end{cases}$$

Note that the first $K-1$ coordinates of these vectors consider times that are each a factor $1+\epsilon$ apart and lie between $d_{\min}$ and $t_{\max}$. Recall that for $t \geq t_{\max}$, it holds that $dbf^*_\tau(t) = u_\tau t$ for each task $\tau_1, \ldots, \tau_n$. Thus, there is no need to consider additional coordinates. The coordinate K is equal to the utilization and will play a special role in our algorithm.

We note that the vectors w^τ as defined above have the structural property that they have initial coordinates zero (for time points that are smaller than the relative deadline of task τ), then L coordinates that equal $dbf^*_\tau(t_k)/t_k$ for some k (that might all have different values), followed by the remaining entries that all equal the utilization u_τ. This observation is formalized below.

Observation 2. *Define $\tilde{k}_\tau := \lceil \log_{(1+\epsilon)}(d_\tau/d_{\min}) \rceil - 1$. A task τ is associated to a vector w^τ from $[0,1]^K$ such that*

$$w^\tau_k = \begin{cases} 0 & \text{if } k \leq \tilde{k}_\tau, \\ \frac{dbf^*_\tau(t_k)}{t_k} & \text{if } k = \tilde{k}_\tau + 1, \ldots, \tilde{k}_\tau + L, \\ u_\tau & \text{otherwise.} \end{cases} \tag{4.3}$$

In particular, each vector has initial coordinates zero, followed by L entries of arbitrary value, followed by all entries equal to u_τ.

The following theorem connects the vector scheduling problem to the sporadic task system scheduling, and follows directly from Proposition 2, Lemmas 16 and 17 and Observation 1. It states that if we can partition the set of vectors $w^\tau \in W$ into sets $W_1, \ldots W_m$ such that $\| \sum_{w^\tau \in W_i} w^\tau \|_\infty \leq 1+\epsilon$, for all $i \in [m]$, then we can feasibly schedule the corresponding tasks in set $\mathcal{T}_i$ on machine i if this machine has a speed of $(1+\epsilon)^3$. Moreover, if $\mathcal{T}$ can be partitioned into sets $\mathcal{T}_i$ such that each of these can be scheduled on a unit-speed machine, then the corresponding sets of vectors W_i satisfy $\| \sum_{w^\tau \in W_i} w^\tau \|_\infty \leq 1+\epsilon$, for all $i \in [m]$.

Theorem 9. *Define the vectors w^τ as in (4.3). Given is a partition of vectors w^τ into m sets $W_1, \ldots, W_m$ and the corresponding partition of tasks $\tau \in \mathcal{T}$ into m sets $\mathcal{T}_1, \ldots, \mathcal{T}_m$. Then, for all machines i,*

(i) if $\left\| \sum_{w^\tau \in W_i} w^\tau \right\|_\infty \leq \alpha$, then $dbf_{\mathcal{T}_i}(t) \leq (1+\epsilon)^2 \alpha t$ for all $t \geq 0$;
(ii) if $dbf_{\mathcal{T}_i}(t) \leq \alpha t$ for all $t \geq 0$, then $\left\| \sum_{w^\tau \in W_i} w^\tau \right\|_\infty \leq (1+\epsilon)\alpha$.

Thus, for $0 < \epsilon < 1$, a $(1+\epsilon)$-approximation algorithm for vector scheduling implies a $(1+\epsilon)^3 = (1+O(\epsilon))$-approximate feasibility test for partitioned scheduling. The result of Chen and Chakraborty [30] follows directly from this connection, and applying Theorem 8. In the next section we show how the running time can be improved for vector scheduling by exploiting the special structure of the vectors w^τ as described in Observation 2.

4.4 The special-case vector scheduling problem

In this section we develop a substantially faster $(1+\epsilon)$-approximation algorithm for vector scheduling, which is specifically tailored towards vectors described in Observation 2. We combine several techniques from bin packing and vector scheduling and design a *sliding-window* dynamic programming approach.

Recall from Section 4.3 that the dimension of our vectors w^τ was defined as $K = \lceil \log_{(1+\epsilon)} \frac{t_{\max}}{d_{\min}} \rceil = 1 + \lceil \log_{(1+\epsilon)}(d_{\max}/\epsilon^2 d_{\min}) \rceil$ and recall that the number of relevant dbf entries is at most $L = 1 + \lceil \log_{(1+\epsilon)}(1/\epsilon^2) \rceil$. Then, let $C = \left(\left\lceil \frac{8L+19}{\epsilon} \right\rceil \right)^L \left\lceil \frac{K(8L+19)}{\epsilon} \right\rceil$. Given a set of vectors W from $[0,1]^K$ as defined in Observation 2, our algorithm determines $O(m^{O(C)})$ time whether the set of vectors W can be scheduled on m machines such that in every coordinate the load is at most $1+\epsilon$, or whether no assignment exists such that in every coordinate the load is at most 1.

After some rounding of the vectors, the main idea of the algorithm is as follows. First, the vectors are classified according to the position of the first non-zero coordinate, then it is determined how the vectors of one class can possibly be scheduled and finally, the schedules of the different classes are combined into one overall schedule. In order to give a high-level overview of our algorithm in Section 4.4.2 and its details in the subsequent subsections, we first need to introduce some notation and concepts in the following subsection.

4.4.1 Notation and definitions

Given $\epsilon > 0$, let K and L be defined as above. Let $0 < \eta < 1$ be a small constant and define $\delta := \eta/K$.

We associate each task τ to a vector w^τ from $[0,1]^K$ as defined in (4.3). We classify these vectors into several classes depending on the index of the first non-zero coordinate. We call a vector a **t-vector** if its first non-zero coordinate is coordinate t.

A **t-configuration** is an $(L+1)$-tuple $(f_1, \ldots, f_L, f_u)$ with $f_k \in \{0, \eta, 2\eta, \ldots, \eta\lfloor 1/\eta \rfloor, 1\}$, for all $k \in [L]$ and $f_u \in \{0, \delta, 2\delta, \ldots, \delta\lfloor 1/\delta \rfloor, 1\}$. Note that although the tuple itself is independent of t, the definition of conforming to a t-configuration is not. We say that (a set of vectors assigned to) a machine i **conforms to** a t-configuration $f = (f_1, \ldots, f_L, f_u)$ if the contribution to coordinate $t+k-1$ is at most f_k, for all $k \in [L]$, and if the contribution to all coordinates $k \geq t+L$ is at most f_u. As the first L elements in a t-configuration can attain one of $\lceil 1/\eta \rceil$ different values and the last element can attain one of $\lceil 1/\delta \rceil$ different values, the number of different t-configurations, denoted by C, is $C := \left(\left\lceil \frac{1}{\eta} \right\rceil \right)^L \left\lceil \frac{1}{\delta} \right\rceil$.

A **t-profile** Q defines a t-configuration for each machine. Therefore, it can be represented by an m-tuple $Q = \langle q_1, \ldots, q_m \rangle$ where q_i denotes the t-configuration corresponding to machine i. Since all machines are identical and the number of t-configurations is bounded by C, a t-profile can also be represented by a C-tuple $Q = (n_1, \ldots, n_C)$ where n_f denotes the number of machines that conform to configuration f. Since the numbers n_f sum up to m, we find that the number of different t-profiles is at most m^C.

Finally, we define the addition of a t-profile Q and a vector $e = (e_1, \ldots, e_L, e_u) \in [0,1]^{L+1}$, i.e., the addition $Q + e = Q' = \langle q'_1, \ldots, q'_m \rangle$, as the pointwise addition of the vector e to each configuration $q_i \in Q$, i.e., $q'_i = q_i + e$ for all $i \in [m]$.

4.4.2 Overview of the algorithm

Our algorithm, as given in Algorithm 2, determines whether we can feasibly schedule all vectors such that each machine has a load of at most $1 + \epsilon$ in every coordinate. The algorithm first applies two rounding steps (see Steps 2 and 3), to limit the number of different vectors.

In Step 4 of the algorithm, we determine for each $t = 1, \ldots, K$ and t-profile R, whether all t-vectors can be scheduled conforming to R; see Section 4.4.4 for the detailed description of how this is determined.

Once we know, for every t, conforming to which t-profiles the set of all t-vectors can be scheduled, we can determine conforming to which t-profiles all vectors together can be scheduled. To accomplish this, we design a *sliding-window dynamic program* (DP). In the DP table, the entry $T[t, Q]$ is evaluated to true if all $\hat{t}$-vectors, with $\hat{t} < t$, can be combined with all t-vectors to conform to a given t-profile Q. Section 4.4.5 describes this procedure in detail. The final result can then easily be obtained by taking $t = K$ and Q equal to the all-1 profile, i.e., $q_i = \mathbf{1}$ for all i. If $T[K, \mathbf{1}]$ returns true, Algorithm 2 can also be used to find the corresponding partition.

Both Step 4 and Step 5 of Algorithm 2 need to be able to determine whether a t-profile

ALGORITHM 2: Vector scheduling algorithm

1: **Input:** a set W of vectors w^τ as defined in Section 4.3, and $\eta \in [0, 1]$.
2: Define $\delta := \eta / K$.
3: For each vector w^τ round each component w^τ_k down to the nearest power of $\frac{1}{1+\eta}$.
4: Modify each vector (Lemma 18)

$$z^\tau_k := \begin{cases} 0 & \text{if } w^\tau_k \leq \delta \left\| w^\tau \right\|_\infty, \\ w^\tau_k & \text{otherwise.} \end{cases}$$

5: Determine whether all t-vectors can be scheduled conforming to t-profile R, for all possible t-profiles R and all t.
6: Let $T[t, Q]$ be true if all k-vectors with $k \leq t$ can be scheduled conform to t-profile Q, and false otherwise. Determine $T[t, Q]$ for all possible t-profiles Q and all t.
7: Return $T[K, \mathbf{1}]$.

R and a $(t-1)$-profile (or t-profile) S can be combined into a t-profile Q. In Section 4.4.6, we show that this can be determined in advance in $O(m^{O(C)})$ time.

4.4.3 Preprocessing

In the preprocessing part, Steps 2 and 3 of Algorithm 2, the vectors are rounded. First, every element of each vector is rounded down to the nearest power of $1/(1+\eta)$. The second rounding step ensures that the positive values in one vector are not more than a factor $1/\delta$ apart. For all $\tau \in \mathcal{T}$ and all $k \in [K]$

$$z_k^\tau := \begin{cases} 0 & \text{if } w_k^\tau \leq \delta \left\| w^\tau \right\|_\infty, \\ w_k^\tau & \text{otherwise.} \end{cases} \tag{4.4}$$

The same rounding step is applied by Chekuri and Khanna [27]. The following lemma states that by rounding the vectors like this, we do not lose too much. It follows directly from the proof of Lemma 2.1 in [27].

Lemma 18 (Chekuri and Khanna [27])**.** *Given a set W of vectors from $[0,1]^K$, let Z be a modified set of W where we replace each vector w^τ in W with a vector z^τ according to (4.4), with $\delta = \eta/K$. Then, for any subset of vectors $Z' \subseteq Z$ and corresponding subset $W' \subseteq W$, we have $\sum_{w^\tau \in W'} w_k^\tau \leq \sum_{z^\tau \in Z'} z_k^\tau + 3\eta \| \sum_{w \in W'} w \|_\infty$.*

4.4.4 Scheduling t-vectors

In this subsection we determine, for each t, conforming to which t-profiles R the set of t-vectors can be scheduled (Step 4 of Algorithm 2). The result of this is used in Step 5 of the algorithm.

For ease of notation, let us denote the set of all t-vectors z^τ in Z by Z_t. Note that any vector in Z_t has value zero in the first $t-1$ coordinates and coordinates $t+L, \ldots, K$ all have the same value. Therefore, in this subsection we act as if z^τ only has dimension $L+1$.

We define $B[t, R]$ to be true if all t-vectors can be scheduled conforming to t-profile $R + (2\eta, \ldots, 2\eta, (L+1)\delta)$ and false otherwise. For a given t and R, Algorithm 3 gives the algorithm for evaluating $B[t, R]$. The idea is to partition the set Z_t into a set of big vectors

ALGORITHM 3: Scheduling the vectors in Z_t conforming to a t-profile R

1: **Input:** profile R, a set Z_t of vectors $z^\tau \in [0,1]^{L+1}$, $\eta \in [0,1]$ and $\delta := \eta/K$.
2: Let $Z_t^{\text{big}} = \{z^\tau \in Z_t \mid \|z^\tau\|_\infty > \delta\}$ and $Z_t^{\text{small}} = Z_t \setminus Z_t^{\text{big}}$.
3: **for all** t-profiles R^{small} and R^{big} that R can be split into **do**
4: Return `true` iff the following two statements are true:
5: (a) Z_t^{big} can be scheduled conforming to $R^{\text{big}} + (\eta, \ldots, \eta, \delta)$ (Lemma 19);
6: (b) Z_t^{small} can be scheduled conforming to $R^{\text{small}} + (\eta, \ldots, \eta, (L+1)\delta)$ (Lemma 20).
7: **end for**
8: Return `false`.

Z_t^{big}, having $\|z^\tau\|_\infty > \delta$, and a set of small vectors Z_t^{small}. Further, we split the t-profile R into two t-profiles R^{big} and R^{small} corresponding to big vectors and small vectors. If all vectors in Z_t^{big} can be scheduled conforming to t-profile R^{big} and if all vectors in Z_t^{small} can be scheduled conforming to t-profile R^{small} then $B[t, R]$ returns true.

Since the t-configurations are "coarse valued" (all values are multiples of either η or δ), it is not so straightforward to split the t-profile R as it would seem. It could be that a coordinate f_k of the t-configuration can be split into two parts yielding a feasible t-profile R^{big} and a t-profile R^{small}, but not in such a way that the two parts are multiples of η. In that case, the corresponding DP cell is erroneously evaluated to false. To circumvent this issue, an additional small error is allowed. For this reason the vector $(\eta, \ldots, \eta, \delta)$ is added to the t-profile R^{big}.

Note that even then, it is not obvious what the correct splitting of profile R should be. Therefore, all possibilities of profiles R^{big} and R^{small} that R can be split into are tried and in this way we "guess" the correct splitting into R^{big} and R^{small}.

Scheduling big vectors

By definition, we know that the maximum value of the coordinates of a big vector is at least δ. Fixing an arbitrary coordinate c, at most $1/\delta$ vectors having coordinate c as their maximum coordinate can be scheduled on one machine when the load in that coordinate may not exceed 1. This reasoning holds for each of the $L+1$ coordinates, and hence at most $(L+1)/\delta$ big vectors can be scheduled on one machine such that the load of each coordinate may not exceed 1. These big vectors are therefore scheduled using a DP, which we call the *inner DP*.

To determine whether the big vectors in Z_t^{big} can be scheduled conforming to a t-profile R^{big}, note that each non-zero coordinate of a big vector is at least δ^2, due to the rounding in Step 3 of Algorithm 2 and the definition of Z_t^{big}. Furthermore, due to the rounding in Step 2 of Algorithm 2, we know that the number of different values that each coordinate of a big vector can take is at most $1 + \log_{(1+\eta)}(1/\delta^2)$. Therefore, there can be at most $(1 + \log_{(1+\eta)}(1/\delta^2))^{L+1}$ different big vectors.

For given t-profile $R^{\text{big}} = \langle r_1, \ldots, r_m \rangle$, the inner DP computes entries $DP[i, V]$ for $i = 1, \ldots, m$ and $V \subseteq Z_t^{\text{big}}$, where $DP[i, V]$ is true if the vectors $V \subseteq Z_t^{\text{big}}$ can be scheduled on machines $1, \ldots, i$ conforming to t-profile R^{big}, that is, conforming to the corresponding t-configurations $r_1, \ldots, r_i$. The recursion in the DP is as follows.

$$DP[i, V] = \bigvee_{Y \subseteq V : Y \text{ on machine } i \text{ according to } r_i} DP[i-1, V \setminus Y].$$

We take the union over all sets $Y \subseteq V$ such that the vectors z^τ in Y can be scheduled on machine i conforming to the t-configuration r_i of machine i. This condition can easily be checked by validating that $\sum_{z^\tau \in Y} z_k^\tau \leq r_{i,k}$ for all $k \in [L+1]$. (Recall that we are abusing notation here as we assumed in this subsection that the vector z^τ has only dimension $L+1$ and hence z_k^τ can directly be compared with $r_{i,k}$.) The base case $DP[1, V]$, which asks whether the vectors in V can be scheduled to machine 1 conforming to t-configuration r_1, follows similarly. This is checking whether $\sum_{z^\tau \in V} z_k^\tau \leq r_{1,k}$ for all $k \in [L+1]$.

The following lemma states that the inner DP can be run in $O\left(m^{O(\frac{1}{\delta\eta})}\right)$ time.

Lemma 19 ([27]). *Given are $(L+1)$-dimensional* big *vectors Z_t^{big} and a t-profile R^{big}. There is an algorithm that determines in $O\left(m^{O(\frac{1}{\delta\eta})}\right)$ time whether there exists a schedule that conforms to t-profile R^{big} or outputs that there is no schedule conforming to R^{big}.*

Proof. There are at most $(1+\log_{(1+\eta)}(1/\delta^2))^{L+1}$ different big vectors possible and this is asymptotically dominated by $\frac{1}{\delta\eta}$ for small enough δ. Hence, any packing of big vectors can be described as a tuple $(b_1, \ldots, b_{\frac{1}{\delta\eta}})$ where b_v indicates the number of vectors of type v that are packed.

As mentioned before, at most $\frac{L+1}{\delta}$ big vectors can be scheduled on any machine such that the load in each coordinate is not more than 1. Hence, there are at most $\frac{m(L+1)}{\delta}$ big vectors in any feasible instance.

Since the numbers b_v sum up to at most $\frac{m(L+1)}{\delta}$, there are at most $O(m^{O(1/\delta\eta)})$ different states in the dynamic program. The number of states is also an upper bound on the time needed to iterate through all different ways to schedule a single machine. The running time is therefore $O\left(m^{O(\frac{1}{\delta\eta})}\right)$. □

Scheduling small vectors

The feasibility question of scheduling the small vectors in Z_t^{small} conforming to the t-profile R^{small} is resolved by finding and rounding a vertex of an appropriate polytope. Let the t-profile R^{small} be identified by the tuple $\langle r_1, r_2, \ldots, r_m\rangle$ of t-configurations. Moreover, we define variables $x_i^\tau = 1$ if vector z^τ is assigned to machine i and 0 otherwise. Then, the feasibility question can be solved by determining whether or not a feasible solution exists satisfying the following constraints.

$$\sum_{z^\tau \in Z_t^{\text{small}}} x_i^\tau z_k^\tau \leq r_{i,k} \qquad \forall i \in [m],\ k \in [L+1];$$

$$\sum_{i=1}^{m} x_i^\tau = 1 \qquad \forall z^\tau \in Z_t^{\text{small}};$$

$$x_i^\tau \in \{0,1\} \qquad \forall i \in [m],\ z^\tau \in Z_t^{\text{small}}.$$

By relaxing the binary constraints for x_i^τ to $x_i^\tau \in [0,1]$, we obtain a polytope. Let us denote by $n^{\text{small}} = |Z_t^{\text{small}}|$. By classical polyhedral theory [22], we know that in any vertex solution of this polytope, there are at most $m(L+1) + n^{\text{small}}$ positive variables because there are $m(L+1) + n^{\text{small}}$ constraints. We show that this implies that there are at most $m(L+1)$ vectors that are fractionally assigned to machines. Suppose by contradiction that $m(L+1)+1$ vectors are assigned to multiple machines. Then there are $n^{\text{small}} - (m(L+1)+1)$ vectors assigned to precisely one machine and $m(L+1)+1$ vectors assigned to at least two machines, which implies that there are at least $n^{\text{small}} - (m(L+1)+1) + 2m(L+1) + 2 = n^{\text{small}} + m(L+1) + 1$ positive variables, which contradicts the fact that there are at most $n^{\text{small}} + m(L+1)$ positive variables. Hence, it must be the case that there are at most $m(L+1)$ vectors which are fractionally assigned to multiple machines.

We then round such a vertex solution to an integral solution by partitioning all vectors that are assigned to at least two machines arbitrarily into m groups of at most $L+1$ vectors

and assign one such group to each machine. Because the vectors are small and each coordinate is smaller or equal to δ, the extra load for every machine on each coordinate is at most $(L+1)\delta$ and $(L+1)\delta \leq K\delta = \eta$ (note that $L+1 \leq K$ holds only if $d_{\max}/d_{\min} > 1+\epsilon$, which we can safely assume in an arbitrary instance). Since a vertex solution of the polytope can be found by simple linear programming techniques, we have the following lemma.

Lemma 20 ([27])**.** *Given a set Z_t^{small} of $(L+1)$-dimensional* small *vectors and a t-profile R^{small}, there is an algorithm that decides in polynomial time whether there exists a schedule for the vectors Z_t^{small} that conforms to t-profile $R^{small} + (\eta, \ldots, \eta, (L+1)\delta)$ or outputs that there is no schedule conforming to t-profile R^{small}.*

Combining the big and small vectors

The following lemma combines Lemmas 19 and 20, yielding the correctness of Algorithm 3. It additionally states the running time of the algorithm.

Lemma 21. *Given a set Z_t of $(L+1)$-dimensional vectors and a t-profile R, Algorithm 3 decides in $O(m^{O(C)})$ time whether there exists a schedule for the vectors in Z_t conforming to profile $R + (2\eta, \ldots, 2\eta, (L+2)\delta)$ or that there exists no schedule for the vectors in Z_t conforming to profile R.*

Proof. Since the t-profiles are tracked with accuracy η in the first L coordinates and with accuracy δ in their utilization, there are no more than m^C different t-profiles. Therefore, there are at most m^{2C} combinations of R^{small} and R^{big} that the profile R can be split into. Lemma 23, which follows in Section 4.4.6, shows that it takes $O(m^{O(C)})$ time to determine whether t-profile R can be split into two t-profiles R^{small} and R^{big}. Thus, in total, it takes at most $m^{2C}O(m^{O(C)}) = O(m^{O(C)})$ time to split the t-profile R into the t-profiles R^{small} and R^{big}. Solving the linear program in Step 5 takes polynomial time and scheduling the big vectors in Step 4 takes $O\left(m^{O(\frac{1}{\delta\eta})}\right)$ time (Lemma 19). The incurred error follows immediately from Lemma 20 and the addition of $(\eta, \ldots, \eta, \delta)$ to R^{big} in Step 4. □

4.4.5 The sliding-window dynamic program

In this subsection, we introduce a dynamic program to determine whether all $\hat{t}$-vectors with $\hat{t} \leq t$ can be scheduled conforming to t-profile Q. To be precise, we compute a table of entries $T[t, Q]$, where $T[t, Q]$ evaluates to true if all $\hat{t}$-vectors with $\hat{t} \leq t$ can be scheduled conforming to t-profile Q. The dynamic program works in K phases as it moves from the first coordinate to coordinate K. While scheduling all t-vectors in a certain phase t, the DP also looks ahead to the next $L-1$ coordinates and the last utilization coordinate to ensure no conflicts arise in these coordinates. That is, we slide a window covering L coordinates from coordinate 1 to coordinate K in as many phases.

Intuitively, phase t corresponds to scheduling the t-vectors, given a partial schedule for all $\hat{t}$-vectors with $\hat{t} < t$. To determine the value of $T[t, Q]$, we split the t-profile Q into a t-profile R and a $(t-1)$-profile S that capture the division of space per machine and

per coordinate between the t-vectors and the other $\hat{t}$-vectors with $\hat{t} < t$, that are dealt with in a recursive way in the dynamic program.

The recursive formula for T can be computed by considering all possible combinations of t-profiles R and $(t-1)$-profiles S that t-profile Q can be split into, and determining whether or not all t-vectors can be scheduled conforming to R and all other $\hat{t}$-vectors with $\hat{t} < t$ can be scheduled conforming to $S + (\eta, \ldots, \eta, \delta)$. That is, for $t > 1$,

$$T[t, Q] = \bigvee_{(R,S)\in\mathcal{X}(Q)} (B[t, R] \wedge T[t-1, S + (\eta, \ldots, \eta, \delta)]), \tag{4.5}$$

where $\mathcal{X}(Q)$ contains all tuples (R, S) of t-profiles R and $(t-1)$-profiles S that Q can be split into.

Note that since the t-configurations are "coarse valued" (all values are multiples of either η or δ), it is unclear how to split the t-profile Q: perhaps a coordinate f_k of the t-configuration can be split into two parts yielding a feasible t-profile R and a $(t-1)$-profile S, but not in such a way that the two parts are multiples of η. In that case, the corresponding DP cell is erroneously evaluated to false. To circumvent this issue, an additional small error in each phase of the sliding-window DP is allowed. For this reason the vector $(\eta, \ldots, \eta, \delta)$ is added to the $(t-1)$-profile S.

The base case of the recursion is

$$T[1, Q] = B[1, Q]. \tag{4.6}$$

To evaluate the running time of computing $T[K, Q]$, we note that $B[t, R]$ is precomputed, as described in Section 4.4.4, and can be accessed in $O(1)$ time.

Lemma 22. *Let W be a set of vectors w^τ as defined in (4.3) and let $\eta > 0$ be small enough. Algorithm 2 decides in $O(Km^{O(C)})$ time whether there exists a partition of the vectors W into m sets $W_1, \ldots, W_m$ such that $\left\|\sum_{w^\tau\in W_i} w^\tau\right\|_\infty < 1 + (8L+19)\eta$ for all i, or that there does not exist a partition with $\left\|\sum_{w^\tau\in W_i} w^\tau\right\|_\infty \leq 1$.*

Proof. We first focus on the running time of the procedure, which is dominated by Steps 4 and 5. In Lemma 21 in Section 4.4.4, the running time of Step 4 is proven to be $O(m^{O(C)})$. Note that there exist C different t-configurations. As there are at most m machines having a certain t-configuration, there are at most m^C different t-profiles and therefore the dynamic program has $O(Km^C)$ different states. For evaluating a recursive step, each possible split of a t-profile into a new t-profile and a $(t-1)$-profile is considered. Since there exist at most $m^C m^C$ such combinations and splitting takes $O(m^{O(C)})$ time (see Lemma 23 in Section 4.4.6), the total running time is $O(Km^{O(C)})$.

To show correctness, note that in each recursive step an error is introduced due to the addition of a vector $(\eta, \ldots, \eta, \delta)$. Furthermore, in the computation of $B[t, R]$ we also incur an error, as it only decides whether all t-vectors can be scheduled conforming to $R + (2\eta, \ldots, 2\eta, (L+2)\delta)$. Fix any coordinate c on any machine and consider the total error on this coordinate. In each phase t for $c - L \leq t \leq c$ an additional error of $\eta + 2\eta$ is introduced and in each phase $t \leq c - L - 1$ an additional error of $\delta + (L+2)\delta$ by the "utilization" of the vectors assigned in those phases. Therefore, the maximum additional

error is at most $(L+1)3\eta + K(L+3)\delta = (4L+6)\eta$ for each coordinate on any machine, as $\delta = \eta/K$.

Finally, let us consider the error that the rounding procedures might introduce. The first rounding procedure rounds each entry of the vector w^τ down to a power of $\frac{1}{1+\eta}$. The resulting error per coordinate per machine is at most a factor $1+\eta$. The second modifying step rounds w_k^τ down to zero in case $w_k^\tau \leq \delta \|w_\tau\|_\infty$. Lemma 18 shows that the corresponding error for the final schedule amounts to an additional factor of at most 3η per coordinate per machine. As the total error incurred by Step 5 amounts to at most $(4L+6)\eta$, we conclude that the total multiplicative error is bounded by a factor $(1+\eta)(1+(4L+9)\eta) \leq 1+(8L+19)\eta$ for $\eta \leq 1$. □

Theorem 10. *Given $\epsilon > 0$, let $C = \left(\left\lceil \frac{8L+19}{\epsilon} \right\rceil\right)^L \left\lceil \frac{K(8L+19)}{\epsilon} \right\rceil$ where $L = 1+\lceil \log_{(1+\epsilon)}(1/\epsilon^2) \rceil$ and $K = 1+\lceil \log_{(1+\epsilon)}(d_{\max}/\epsilon^2 d_{\min}) \rceil$. Then, given a set of vectors W from $[0,1]^K$ as defined in Observation 2, Algorithm 2 determines in $O(m^{O(C)})$ time whether the set of vectors W can be scheduled on m machines such that in every coordinate the load is at most $1+\epsilon$, or whether no feasible assignment exists.*

Proof. The theorem follows directly from Lemma 22. Setting $\eta := \epsilon/(8L+19)$ in this lemma leads to a schedule with height at most $1+(8L+19)\eta = 1+\epsilon$. □

4.4.6 Splitting t-profiles

Recall that a t-configuration $f = (f_1, \ldots, f_L, f_u)$ specifies how much space might be used by the vectors not scheduled yet on a certain machine in the coordinate $k \in \{t, \ldots, t+L-1\}$ and in coordinate K. In the inner DP we want to split a t-configuration into two other t-configurations, one specifying the space for the big t-vectors and the other specifying the space for the small t-vectors. In the sliding-window DP on the other hand, we want to split a t-configuration into another t-configuration and a $(t-1)$-configuration, specifying the space the vectors in Z_t might use and the space the vectors in $\cup_{\hat{t}=1}^{t-1} Z_{\hat{t}}$ might use, respectively, per machine per relevant coordinate.

A t-configuration $f = (f_1, \ldots, f_L, f_u)$ can be split into a t-configuration g and a t-configuration h if and only if

- $f_k \geq g_k + h_k$ for all $1 \leq k \leq L$;
- and $f_u \geq g_u + h_u$.

A t-configuration f can be split into a t-configuration g and a $(t-1)$-configuration h if and only if

- $f_k \geq g_k + h_{k+1}$ for all $1 \leq k \leq L-1$;
- and $f_L \geq g_L + h_u$;
- and $f_u \geq g_u + h_u$.

Splitting of t-profiles is done in a similar way. A t-profile $Q = \langle q_1, \ldots, q_m \rangle$ can be split into a t-profile $R = \langle r_1, \ldots, r_m \rangle$ and a t-profile $S = \langle s_1, \ldots, s_m \rangle$ if and only if there are permutations $\alpha, \beta : [m] \to [m]$ such that the t-configuration q_i can be split into the t-configuration $r_{\alpha(i)}$ and t-configuration $s_{\beta(i)}$. In the same way a t-profile can be split into a t-profile and a $(t-1)$-profile.

Lemma 23. *Determining whether a t-profile Q can be split into a t-profile R and a $(t-1)$-profile S can be done in $O(m^{O(C)})$ time.*

Proof. The goal is to pair configurations from R and configurations from S such that each pair can be combined into a unique configuration from Q. We use the fact that there are at most m^C different profiles because there are C different configurations and a profile can be alternatively denoted as the tuple $(n_1, \ldots, n_C)$ where n_f is the number of machines conforming to configuration f.

In a simple dynamic programming approach we can keep track of configurations not yet paired. This gives at most $m^{O(C)}$ states. In each state we need to find a configuration from Q that can be split into a configuration from R and a configuration from S, and check whether the remaining configurations can be matched. This can be done in $O(m^{O(C)})$ time for each state by considering all possibilities. This lead to an algorithm that runs in $O(m^{O(C)})$ time. □

4.5 Conclusion

Combining all ingredients from the previous sections yields our main theorem.

Theorem 11. *Given $\epsilon > 0$, a task set $\mathcal{T}$ and m parallel identical processors, there is an algorithm which correctly decides in $O\left(m^{O(f(\epsilon)\log\lambda)}\right)$ time whether $\mathcal{T}$ can be feasibly partitioned with a speedup of $1+\epsilon$, or no feasible partition exists in case the machines run at unit speed, where $\lambda = d_{\max}/d_{\min}$, the ratio between the largest and smallest deadline, and $f(\epsilon)$ is a function depending solely on ϵ.*

Proof. Theorem 10 states that in $O(m^{O(C)})$ time it can be determined whether a feasible solution to the vector scheduling problem exists with a speedup factor of $1+\epsilon$, or whether no such partition of the vectors over the machines exists without speedup. Thus in light of Theorem 10, Theorem 9 implies that if there exists a feasible partition for the vector scheduling problem, then this partition is feasible for our real-time scheduling problem if the machines receive a speedup factor of $(1+\epsilon)^3$, and that if no feasible partition for the vector scheduling problem exists, then no feasible partition exists for the real-time scheduling problem in case the machines run at speed $1/(1+\epsilon)$. Rescaling ϵ appropriately yields the stated result. □

Chapter 5

Mixed-criticality scheduling

Most results in this chapter previously appeared in [14] and [15].

5.1 Introduction

Work in the field of mixed-criticality scheduling is highly motivated from embedded systems where multiple functionalities are implemented on a shared computing device. In safety-critical systems, it is often the case that not all functionalities implemented have the same importance (or criticality) to the overall system. Such systems are called mixed-criticality (MC) systems. We start with an example, taken from the domain of unmanned aerial vehicles (UAVs).

UAVs are aircrafts without a human pilot on board. Its flight may be controlled remotely by a pilot on the ground, or autonomously by computers in the vehicle. UAVs (or drones, as they are called as well) are mainly used for military purposes, for example for reconnaissance and surveillance operations. The functionalities on board such a vehicle can be divided into two classes. There is the class of mission-related functionalities, like capturing images and transmitting them further. Then there are the functionalities related to a safe flight of the vehicle. The flight-related functionalities are subject to mandatory certification by statutory certification authorities (CAs) to obtain permission to operate over civilian airspace. These CAs are not interested in the mission-related functionalities. For the flight-related functionalities, however, they need a very high level of certainty about schedulability of all computational tasks. The manufacturer (and user) of the vehicle on the other hand, are interested in both types of functionalities, but are typically satisfied with a lower level of assurance on the flight-related functionalities.

This reasoning gives us a two-level mixed-criticality system, where the two types of functionalities are in two levels of criticality.

Level 1: the mission-critical functionalities, concerning reconnaissance and surveillance;
Level 2: the flight-critical functionalities, concerning a safe operation of the aircraft.

In the certification process, the CA makes certain assumptions about the worst-case run-time behavior of the system. We focus here on one aspect of run-time behavior: the worst-case execution time (WCET) of pieces of code. Determining the exact WCET of

an arbitrary piece of code is intractable, but system engineers use tools and techniques to determine upper bounds on them. As the CAs are seeking a higher level of correctness, they will choose to use more conservative tools for determining WCET bounds than the manufacturer does. This will typically result in higher WCET bounds for validating the higher-critical functionalities.

Finding procedures that certify efficiently such mixed-criticality systems has been identified as a challenging collection of problems [12]. In Section 5.2 we will define the scheduling problem that we are interested in formally, but we will sketch it here already, using the example given above. Functionalities that are of criticality level 1 are only interesting to the manufacturer, while the level-2 functionalities are interesting to both the manufacturer and the certification authority, but the latter will have a higher WCET estimate for them than the former. The level-2 tasks will therefore have two WCETs parameters, one in the lower criticality level (level 1—relevant to the manufacturer) and one in the higher criticality level (level 2—relevant to the CA). The scheduling goal is to find a schedule such that the following requirements are satisfied:

- (*Validation by manufacturer:*) if all execution times are no larger than the level-1 WCETs, it is required that all jobs meet their deadlines.
- (*Certification by CA:*) if the level-2 jobs have execution times that are larger than the level-1 WCETs, but no larger than the level-2 WCETs, only all level-2 critical jobs are required to complete by their deadlines, disregarding the level-1 functionalities.

The difficulty of finding a schedule that meets these requirements is that scheduling decisions must be made prior to run time. The actual execution times will only be learned by executing the jobs, until they signal that they have completed.

Note that while in the example only two criticality levels are specified, generally more criticality levels can be present. In Section 5.3 we will give results for a system consisting of K criticality levels, where K is an arbitrary integer.

5.1.1 Related work

Based on the observation that although the CAs need to use very conservative tools for determining WCETs, less conservative tools should suffice for validating less critical functionalities, Vestal [71] was the first to propose that different WCET values be used in different criticality levels. He studied the fixed-priority scheduling of mixed-criticality sporadic task systems on a single preemptive processor, but gives only empirical results.

Baruah et al. [13] show that schedulability analysis of MC systems is strongly NP-hard. This continues to hold, even in a two-level instance where all jobs arrive simultaneously. In [20] the scheduling of a finite collection of independent jobs in a two-level MC system on a single processor was studied. The authors analyze the effectiveness of two techniques widely used in practice in real-time scheduling: the reservations-based approach and the priority-based approach. They propose an algorithm called Own Criticality Based Priority (OCBP) and show that this has a processor speedup factor equal to the golden ratio ϕ. Later, these results were extended to tight bounds for any number of criticality levels [13].

The mixed-criticality model has been extended to recurrent task systems in [55]. For a two-level mixed-criticality task system, the authors give a scheduling policy and a pseudo-polynomial-time schedulability test for a single preemptive processor with a speedup bound of ϕ. Guan et al. [43] subsequently proposed an algorithm called PLRS that only has quadratic run-time complexity, able to schedule a wider range of instances.

The algorithm EDF-VD, that will be presented in Section 5.3, was introduced in [15] and the performance (for two-level systems only) was improved in [14]. The idea of EDF-VD is that higher-criticality tasks have their deadlines reduced as long as the system is in lower criticality levels, to ensure schedulability across a criticality change. Others have proposed different algorithms for finding the adjusted deadlines.

The approach of Ekberg and Yi [38, 39] is that for each task separately (both low- and high-criticality tasks) they consider to what extent its deadline can be lowered to shape the demand bound function differently. Since it would require too much computation to consider all possibilities, a pseudo-polynomial-time heuristic is proposed that is essentially greedy. The downside of their more general approach is that no worst-case guarantees can be given and the authors' claim that their test outperforms EDF-VD is only supported by experimental results. A beautiful contribution of [39] concerns a generalization of the mixed-criticality task model to models where not only the execution time changes in a criticality change, but all task parameters are allowed to change.

Easwaran [34] introduces a different technique for determining the virtual deadlines, that also tries to decrease the deadlines of the high-criticality tasks separately. This technique combined with his new schedulability test seems to be able to schedule a larger fraction of randomly generated instances than the algorithm of Ekberg and Yi.

A different approach to using spare capacity is derived by Su and Zhu [68] by exploiting *elastic scheduling*. A minimum service requirement is defined for low-criticality tasks, expressed by a maximum period. The system should be schedulable if the high-criticality tasks have a high-criticality execution time and for the low-criticality tasks this maximum period is considered. If there is some slack because high-criticality tasks use less than their high-criticality execution time, the low-criticality tasks can have smaller periods and thus run more frequently. Simulation results again show that (for certain parameters settings) an improvement over EDF-VD can be obtained, at the cost of higher overhead.

See [25] for an extensive survey of the research that has been conducted within the real-time scheduling community on mixed-criticality scheduling problems.

5.1.2 Our results

In Section 5.3 we present a schedulability test, combined with a scheduling policy, called EDF-VD, designed for scheduling implicit-deadline MC task systems on a single preemptive processor. The test can be applied to MC task systems with K levels, thereby extending the result from [55]. For 2-level and 3-level systems we subsequently give speedup bounds. We show that any 2-level (respectively, 3-level) implicit-deadline task system that can be scheduled by a clairvoyant[1] algorithm on a given processor, can be scheduled by EDF-VD on a processor that is 4/3 (respectively, 2) times as fast. Finally we show

[1]See Section 5.2 for the definition of clairvoyance.

that no non-clairvoyant algorithm can guarantee correctness on a processor that is less than 4/3 times as fast as the processor available to the clairvoyant algorithm. Hereby, we show that for an implicit-deadline dual-criticality system, EDF-VD is optimal from the processor speedup viewpoint.

Whereas EDF-VD is a policy with dynamic priorities, in Section 5.4 we study scheduling implicit-deadline tasks with a fixed-priority policy. Although fixed-priority policies are more restrictive with respect to the processor utilization of tasks systems they can handle, they can be preferred over dynamic priorities due to ease of implementation. The priority list is determined beforehand and needs no updating during the scheduling process, while the priority list needs constant updating in case of dynamic priorities. For a dual-criticality system we show that our fixed-priority algorithm RM-VP can schedule any task system that is clairvoyantly feasible on a unit-speed processor, on a processor running at speed $\phi/\ln 2 \approx 2.334$. Further we show that no fixed-priority scheduling policy can schedule a task system that is clairvoyantly feasible on a unit-speed machine on a processor that has speed lower than 2.

Section 5.5 considers a finite collection of independent jobs, to be scheduled on multiple identical processors. The OCBP scheduling procedure from [13] is extended to dual-criticality systems on multiple machines. We show that any system that is clairvoyantly feasible on m processors running at unit speed, is schedulable by OCBP on m processors that are $\phi + 1 - \frac{1}{m}$ times as fast.

Before giving these results, notation and concepts needed are formally introduced in Section 5.2.

5.2 Preliminaries

Let $K \geq 1$ be an integer, denoting the number of criticality levels. We will formally define MC jobs and MC tasks and give some definitions on feasibility of MC systems.

MC jobs A job in a K-level MC system is characterized by a 4-tuple of parameters: $J_j = (\chi_j, a_j, d_j, c_j)$, where

- χ_j is the job's criticality level in $\{1, \ldots, K\}$;
- a_j is the arrival time of the job;
- d_j is the deadline (where $d_j \geq a_j$); and
- c_j is a vector $(c_j(1), c_j(2), \ldots, c_j(K))$ of WCETs, one for each criticality level up to K. It is assumed that $c_j(1) \leq c_j(2) \leq \cdots \leq c_j(\chi_j)$ and $c_j(k) = c_j(\chi_j)$, for each $k > \chi_j$.

Each job J_j in a collection $J_1, \ldots, J_n$ should receive execution time γ_j (where $\gamma_j \in [0, c_j(\chi_j)]$), within time window $[a_j, d_j]$. The value of γ_j is not known in advance, but is discovered by executing job J_j until it signals completion. A collection of realized values $(\gamma_1, \gamma_2, \ldots, \gamma_n)$ is called a scenario. The criticality level of a scenario $(\gamma_1, \ldots, \gamma_n)$ is defined as the smallest integer k such that $\gamma_j \leq c_j(k)$ for each job J_j. The crucial aspect of the model is that, in a scenario of level k, it is necessary to guarantee only that jobs of

criticality at least k are completed before their deadlines. In other words, once a scenario is known to be of level k, the jobs of criticality $k-1$ or less can be safely dropped.

MC task systems Let $\mathcal{T} = (\tau_1, \ldots, \tau_n)$ be a system of n tasks, where each task τ_i releases a possibly infinite sequence of MC jobs. Each task τ_i is characterized by a tuple (χ_i, d_i, p_i, c_i), where

- the criticality level of the task is χ_i;
- d_i is the deadline, relative to the arrival of a job;
- the period p_i denotes the minimum interarrival time between two consecutive jobs of task τ_i; and
- the vector $c_i = (c_i(1), \ldots, c_i(K))$ denotes the WCETs at all criticality levels up to K. Again, $c_i(1) \leq c_i(2) \leq \cdots \leq c_i(\chi_i)$ and $c_i(k) = c_i(\chi_i)$, for each $k > \chi_i$.

The jobs generated by task τ_i are denoted as $(J_{i1}, J_{i2}, \ldots)$. An MC job J_{ij} generated by τ_i is defined by the parameters of τ_i and the parameters (a_{ij}, γ_{ij}), where

- a_{ij} is the arrival time of the job (such that $a_{i,j+1} \geq a_{ij} + p_i$ for all J_{ij});
- $\gamma_{ij} \in [0, c_i(\chi_i)]$ is the execution requirement of job J_{ij} and is only discovered by execution of J_{ij}; and
- the absolute deadline d_{ij} of J_{ij} is defined as $d_{ij} = a_{ij} + d_i$.

A collection $I = (a_{ij}, \gamma_{ij})_{i \in [n], j \geq 1}$ of arrival times and execution requirements now constitutes a scenario. The criticality level of a scenario is defined as the smallest integer k such that $\gamma_{ij} \leq c_i(k)$ for all jobs J_{ij}. Such an integer always exists since γ_{ij} is assumed to be at most $c_i(\chi_i)$. As before, in a scenario of level k, only jobs from tasks of criticality at least k need to meet their deadlines and jobs from tasks with criticality strictly less than k can be omitted.

In the remainder of this section we will give some definitions that are formulated for task systems. The definitions also hold for a collection of independent jobs. Just note that there will be only one job J_{i1} per "task" and their arrival times are known. So the differences between scenarios will only be in the realized execution times.

Definition 4. *Given a scenario generated by task system $\mathcal{T}$, a schedule for it is called* feasible *if it schedules every job such that, if the level of the scenario is k, every job J_{ij} with $\chi_i \geq k$ receives execution time γ_{ij} between its arrival time a_{ij} and deadline d_{ij}. A task system $\mathcal{T}$ is* (clairvoyantly) feasible *if for every possible scenario a feasible schedule exists.*

An *online* (or *non-clairvoyant*) scheduling policy for an MC task system discovers the criticality of a scenario only by executing the jobs. Scheduling decisions can be based only on the partial information revealed thus far.

Definition 5. *An online scheduling policy A is* correct *for a feasible task system $\mathcal{T}$ if for every scenario of $\mathcal{T}$ the policy generates a feasible schedule.*

Definition 6. *A mixed-criticality task system $\mathcal{T}$ is called* MC-schedulable *if it admits some correct scheduling policy. Note that MC-schedulability implies clairvoyant feasibility.*

In later parts of this chapter, we want to assess the quality of scheduling policies by calculating the required speedup factor to schedule a task set, given that it is MC-schedulable. However, since we cannot determine MC-schedulability in a different way than by giving a correct scheduling policy, and since MC-schedulability implies clairvoyant schedulability, in fact we compare to the conditions given for clairvoyant schedulability.

5.3 Implicit-deadline tasks on a single processor

In this section we study scheduling implicit-deadline task systems on a single processor. Recall that in implicit-deadline systems $d_i = p_i$ for all $\tau_i \in \mathcal{T}$. This also means that the parameter vector characterizing such tasks will now be of the form $\tau_i = (\chi_i, p_i, c_i)$.

For MC tasks we need to extend the notion of utilization (see Chapter 1) to capture the different execution time parameters at different criticality levels. We define the utilization of task τ_i at level k as

$$u_i(k) = \frac{c_i(k)}{p_i}, \qquad \text{for } i = 1, \ldots, n, \text{ for } k = 1, \ldots, \chi_i.$$

The total utilization at level k of tasks with criticality level ℓ is then defined as

$$U_\ell(k) = \sum_{\tau_i : \chi_i = \ell} u_i(k), \qquad \text{for } \ell = 1, \ldots, K, \text{ for } k = 1, \ldots, \ell.$$

Proposition 1 gives a schedulability condition for an implicit-deadline task system that consists of one criticality level. Reading $\sum_{\tau_i \in \mathcal{T}} u_i(1)$ instead of $\sum_{\tau \in \mathcal{T}} u_\tau$, it tells us that a 1-level task system is feasible on a speed-σ processor if and only if $U_1(1) \leq \sigma$. In a feasible instance of K levels, this condition holds for any level. This yields the following *necessary* condition for feasibility in mixed-criticality systems.

Proposition 10. *If $\mathcal{T}$ is feasible on a unit-speed processor, then*

$$\max_{k=1,\ldots,K} \sum_{\ell=k}^{K} U_\ell(k) \leq 1. \tag{5.1}$$

Proof. For each $k = 1, \ldots, K$, consider a scenario where each task τ_i with $\chi_i \geq k$ releases jobs with execution requirement $c_i(k)$. □

In the presence of multiple criticality levels, however, EDF does not necessarily produce a feasible schedule, even if the utilization in each level is less than 1, i.e., if the necessary conditions given above are satisfied. Consider the following example.

Example 3. *Consider a task system $\mathcal{T} = (\tau_1, \tau_2)$ with the following parameters:*

τ_i	χ_i	p_i	$c_i(1)$	$c_i(2)$
τ_1	1	4	2	2
τ_2	2	6	1	5

Note that the total utilization at level 1 is $U_1(1) + U_2(1) = 2/3$*, while at level 2 it is* $U_2(2) = 5/6$*. However, EDF may fail to meet deadlines, as follows. Assume jobs are released as early as possible. At time 0, EDF schedules the first job of* τ_1*, since that has the earliest deadline. The job finishes at time 2 and EDF starts running the first job of* τ_2*. If it turns out that this job exhibits level-2 behavior, it will execute for 5 time units and it will miss its deadline at time 6. If we had started the other way around, either the first job of* τ_2 *would have finished after 1 time unit and there would have been enough time to schedule the job from* τ_1 *before its deadline at time 4, or the scenario would have exhibited level-2 behavior and we could have discarded the job from* τ_1*.*

The scheduling algorithm that we propose is called EDF with Virtual Deadlines (EDF-VD) and is an adaptation of EDF that handles the problem sketched above, while maintaining some of the desirable properties of EDF. In the remainder of this section, we first introduce the algorithm EDF-VD and give a corresponding schedulability condition for a system of K criticality levels. Then, for the cases of 2 and 3 levels, we assess the quality of the algorithm by calculating the required speedups.

5.3.1 Overview of EDF-VD

We denote by $\mathcal{T}$ the MC implicit-deadline task system that we want to schedule on a unit-speed preemptive processor. EDF-VD performs a schedulability test prior to run time, to determine whether or not $\mathcal{T}$ can be feasibly scheduled on the processor. If the task system is deemed schedulable, the schedulability test also provides us with an integer parameter k (with $1 \leq k \leq K$) that will serve as an input for the scheduling algorithm and, for each task $\tau_i \in \mathcal{T}$, with a parameter that we call a *virtual (relative) deadline*, denoted $\hat{d}_i$, which is never larger than d_i. The algorithm for computing these parameters is described in pseudo-code form in Algorithm 4. The details of this algorithm are best understood after reading Section 5.3.2, where the correctness of the algorithm is proved.

Run-time scheduling in EDF-VD consists of K variants, called EDF-VD(1), EDF-VD(2), ..., EDF-VD(K). Each of these is related to a different value of the parameter that was provided by the schedulability conditions. If the value of the parameter returned by Algorithm 4 is k, then the corresponding variant EDF-VD(k) is applied.

Algorithm EDF-VD(k) While the system is in level $1, 2, \ldots, k$ all tasks have a *virtual (relative) deadline* $\hat{d}_i$ assigned. For tasks of criticality level k or below this virtual deadline will be equal to the original deadline, but for tasks of criticality level $k+1, \ldots, K$, their deadline is scaled by a factor $x \leq 1$. The tasks are then scheduled according to EDF with respect to the virtual deadlines. As soon as the system reaches criticality level $k+1$, all jobs from tasks of criticality k or below have been discarded and for all other tasks, the original deadlines are restored. Algorithm 5 gives a pseudo-code describing this algorithm. The function `current_level()` returns the level exhibited by the scenario so far, that is,

$$\texttt{current\_level()} = \min\{l : \tilde{\gamma}_{ij} \leq c_i(l) \text{ for all jobs } J_{ij} \text{ (partially) scheduled so far}\},$$

where $\tilde{\gamma}_{ij}$ is the part of γ_{ij} that has been observed thus far. Note that if $k = K$ then Algorithm 5 is simply EDF, as no scaling of the deadlines occurs in Algorithm 4.

5.3.2 Schedulability conditions

Theorem 12. *Given an implicit-deadline task system $\mathcal{T}$, if either*

$$\sum_{\ell=1}^{K} U_\ell(\ell) \leq 1 \tag{5.2}$$

or, for some k $(1 \leq k < K)$, the following condition holds:

$$1 - \sum_{\ell=1}^{k} U_\ell(\ell) > 0 \quad \text{and} \quad \frac{\sum_{\ell=k+1}^{K} U_\ell(k)}{1 - \sum_{\ell=1}^{k} U_\ell(\ell)} \leq \frac{1 - \sum_{\ell=k+1}^{K} U_\ell(\ell)}{\sum_{\ell=1}^{k} U_\ell(\ell)}, \tag{5.3}$$

then $\mathcal{T}$ can be correctly scheduled by EDF-VD.

Proof. If (5.2) holds, then all jobs can be scheduled for their worst-case execution time at their own criticality level. Hence EDF-VD(K), that is, EDF without deadline scaling, will

ALGORITHM 4: EDF with Virtual Deadlines (EDF-VD) – Offline preprocessing phase

Input: task system $\mathcal{T} = (\tau_1, \ldots, \tau_n)$ to be scheduled on a unit-speed preemptive processor

if $\sum_{l=1}^{K} U_l(l) \leq 1$ **then**
  $k \leftarrow K$
  for $i = 1, 2, \ldots, n$ **do**
    $\hat{d}_i \leftarrow d_i$
  end for
else
  Let k $(1 \leq k < K)$ be such that (5.3) holds
  if no such k exists **then**
    return unschedulable
  else
    Let $x \in (0, 1]$ be such that (5.6) and (5.7) hold
    for $i = 1, 2, \ldots, n$ **do**
      if $\chi_i \leq k$ **then**
        $\hat{d}_i \leftarrow d_i$
      else
        $\hat{d}_i \leftarrow x d_i$
      end if
    end for
  end if
end if
return (schedulable, k, $(\hat{d}_i)_{i=1}^{n}$)

yield a correct schedule. Therefore, from here on the proof focuses on the case that (5.2) does not hold and (5.3) holds for some $k < K$.

The proof consists of two steps. In the first step, we show that if there is a scaling parameter $x \leq 1$ such that the following two inequalities hold

$$\sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} \frac{U_\ell(k)}{x} \leq 1, \tag{5.4}$$

$$x \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} U_\ell(\ell) \leq 1, \tag{5.5}$$

then EDF-VD(k) is a correct scheduling policy for $\mathcal{T}$.

In the second step, we show that if (5.3) holds for some $k < K$, then there exists $x \leq 1$ such that (5.4) and (5.5) hold. Further, we show how to find such x.

For the first step, we determine an upper bound on the "virtual utilization" as long as the virtual deadlines are applied, if EDF-VD(k) is our scheduling algorithm. We will slightly abuse notation and write $\hat{p}_i$ for the "virtual" period that equals the virtual deadline $\hat{d}_i$ for all tasks τ_i, in order to define the "virtual" utilization. Assume for all tasks τ_i with χ_i at most k, that all jobs are executed at their own-criticality execution requirement. For all tasks τ_i with $\chi_i > k$, the jobs are executed at criticality level k. Note that by this assumption, no complications can occur at criticality changes before level $k+1$ is reached. The "virtual utilization" of this tasks system in level $1, 2, \ldots, k$ equals

$$\begin{aligned}
\hat{U}(k) :=& \sum_{\tau_i} \frac{c_i(k)}{\hat{p}_i} = \sum_{\tau_i} \frac{c_i(k)}{\hat{d}_i} \\
=& \sum_{\tau_i:\, \chi_i \leq k} \frac{c_i(\chi_i)}{d_i} + \sum_{\tau_i:\, \chi_i > k} \frac{c_i(k)}{x d_i} \\
=& \sum_{\tau_i:\, \chi_i \leq k} \frac{c_i(\chi_i)}{p_i} + \sum_{\tau_i:\, \chi_i > k} \frac{c_i(k)}{x p_i} \\
=& \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} \frac{U_\ell(k)}{x}.
\end{aligned}$$

Note that a sufficient condition for correctness in levels $1, 2, \ldots, k$ is that $\hat{U}(k) \leq 1$. Hence, we find that (5.4) ensures correctness in levels $1, 2, \ldots, k$.

For any level $\ell > k$, assume by contradiction that a deadline miss occurs in some scenario. We bound the execution requirement of all tasks until the time of the first deadline miss, which is denoted by t_f. Let I denote a minimal collection of jobs released by $\mathcal{T}$ on which a deadline is missed (by *minimal*, we mean that EDF-VD would meet all deadlines if scheduling any proper subset of I). Without loss of generality we assume that the first job arrival is at time zero, and further that (5.4) and (5.5) hold. Hence the job that misses a deadline is of level $\ell > k$, because at time t_f the system is at least in level $k+1$. Let t^* denote the time where behavior of level $k+1$ is first seen and the system transfers to level $k+1$.

Fact 1. *All jobs receiving execution in $[t^*, t_f)$ have deadline at most t_f.*

Proof. Suppose there is a job that has deadline larger than t_f and receives some execution between t^* and t_f, say in the interval $[t_1, t_2)$. This means that during $[t_1, t_2)$ there are no jobs pending with deadline at most t_f. Then, the set of jobs obtained by considering only jobs with arrival time at least t_2 will also miss a deadline at t_f, which contradicts the assumed minimality of I. □

We define the quantity $\omega_i(t)$ as the cumulative execution requirement of jobs of task τ_i until time t and derive upper bounds for this quantity for all tasks. Among all jobs executing in $[t^*, t_f)$, let J_0 be the job with the earliest arrival time. Denote by a_0 its arrival time and by d_0 its absolute deadline.

Fact 2. *For any task τ_i having $\chi_i \leq k$, it holds that*

$$\omega_i(t_f) \leq \big(a_0 + x(t_f - a_0)\big)u_i(\chi_i).$$

Proof. Note that no job of τ_i will receive execution after t^*. If such a job executes after a_0, it must have a deadline no larger than the virtual deadline of J_0, which is $a_0 + x(d_0 - a_0)$. Since $t_f \geq d_0$ by Fact 1, this means that no job of task τ_i with deadline greater than $a_0 + x(t_f - a_0)$ will execute after a_0.

ALGORITHM 5: EDF with Virtual Deadlines (EDF-VD) – Run-time dispatching

---

Input: task system $\mathcal{T} = (\tau_1, \ldots, \tau_n)$, integer k $(1 \leq k \leq K)$, modified deadlines $(\hat{d}_i)_{i=1}^n$

loop
  on job arrival:
  if a job of task τ_i arrives at time t, assign it a virtual deadline equal to $t + \hat{d}_i$
  on job arrival/completion:
  schedule the active job, among the tasks τ_i such that $\chi_i \geq$ `current_level`(), having earliest absolute virtual deadline (ties broken arbitrarily)
  on `current_level`() $> k$:
  break from the loop
end loop

`/* schedule the active job, among the tasks` τ_i `such that` $\chi_i > k$`, having earliest absolute deadline (ties broken arbitrarily) */`

loop
  on job arrival:
  if a job of task τ_i arrives at time t, assign it a deadline equal to $t + d_i$
  on job arrival/completion:
  schedule the active job, among the tasks τ_i such that $\chi_i \geq$ `current_level`(), having earliest absolute deadline (ties broken arbitrarily)
end loop

Suppose now that a job with $\chi_i \leq k$ and deadline larger than $a_0 + x(t_f - a_0)$ was executed for some time before a_0. Let t_2 denote the latest instant at which any such job executes. This means that at this instant, there were no jobs with absolute deadline at most $a_0+x(t_f-a_0)$ awaiting execution. Hence, the set of jobs obtained by considering only those jobs in I that have arrival time at least t_2 also misses a deadline. This contradicts the assumed minimality of I.

This implies that there are at most $(a_0 + x(t_f - a_0))/p_i$ jobs of τ_i until time t_f; each of them requires an execution time of at most $c_i(\chi_i)$. Therefore, the total execution requirement of τ_i is bounded by $\big(a_0 + x(t_f - a_0)\big)u_i(\chi_i)$. □

Fact 3. *Any task τ_i with $\chi_i > k$ has*

$$\omega_i(t_f) \leq \frac{a_0}{x}u_i(k) + (t_f - a_0)u_i(\chi_i).$$

Proof. We distinguish two cases.
Case 1: Task τ_i does not release a job at or after a_0. Note that each job of τ_i has a virtual deadline of at most $a_0 + x(t_f - a_0)$. To show that this is true, suppose there was a job with a larger virtual deadline and let t_2 denote the latest time instant at which this job executes. The job sequence consisting of only those jobs with an arrival time larger than t_2 also misses a deadline and this is in contradiction with the assumed minimality of I.

Hence, each job of τ_i has an actual deadline of at most $a_0/x + t_f - a_0$ and there are at most $\frac{a_0/x+t_f-a_0}{p_i}$ of them. Since these jobs do not execute in $[t^*, t_f)$ (else, J_0 would not be the one with earliest release among those), the execution requirement per job is at most $c_i(k)$. Combining these observations, we bound the execution requirement of jobs from task τ_i by

$$\begin{aligned}\Big(\frac{a_0}{x} + t_f - a_0\Big)\frac{c_i(k)}{p_i} &= \frac{a_0}{x}u_i(k) + (t_f - a_0)u_i(k)\\ &\leq \frac{a_0}{x}u_i(k) + (t_f - a_0)u_i(\chi_i).\end{aligned}$$

Case 2: Task τ_i releases one or more jobs at or after a_0. Let a_i denote the first release of a job from τ_i greater than or equal to a_0. The previously released job of τ_i did not execute in $[t^*, t_f)$, by definition of a_i and a_0. Therefore, it is safe to assume that until a_i the execution requirement per job from τ_i was bounded by $c_i(k)$ and after that it is bounded by $c_i(\chi_i)$. Hence, the cumulative requirement of all jobs from τ_i is bounded by

$$\begin{aligned}a_iu_i(k) + (t_f - a_i)u_i(\chi_i) &\leq a_0u_i(k) + (t_f - a_0)u_i(\chi_i)\\ &\leq \frac{a_0}{x}u_i(k) + (t_f - a_0)u_i(\chi_i),\end{aligned}$$

where the first inequality comes from the facts that $a_0 \leq a_i$ and $u_i(k) \leq u_i(\chi_i)$ and the second one from $x \leq 1$. □

Summing the cumulative requirements over all tasks gives

$$\begin{aligned}
&\sum_{i:\chi_i \le k} \omega_i(t_f) + \sum_{i:\chi_i > k} \omega_i(t_f) \\
&\qquad \le \sum_{i:\chi_i \le k} \big(a_0 + x(t_f - a_0)\big) u_i(\chi_i) + \sum_{i:\chi_i > k} \Big(\frac{a_0}{x} u_i(k) + (t_f - a_0) u_i(\chi_i)\Big) \\
&\qquad = a_0 \Bigg(\sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} \frac{U_\ell(k)}{x} \Bigg) + (t_f - a_0) \Bigg(x \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} U_\ell(\ell) \Bigg) \\
&\qquad \le a_0 + (t_f - a_0) \Bigg(x \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} U_\ell(\ell) \Bigg),
\end{aligned}$$

where the last inequality comes from the assumption that (5.4) holds.

The assumed deadline miss implies

$$\begin{aligned}
& a_0 + (t_f - a_0) \Bigg(x \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} U_\ell(\ell) \Bigg) > t_f \\
\Leftrightarrow \quad & (t_f - a_0) \Bigg(x \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} U_\ell(\ell) \Bigg) > t_f - a_0 \\
\Leftrightarrow \quad & x \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} U_\ell(\ell) > 1,
\end{aligned}$$

but this directly contradicts (5.5).

Now that we have shown that (5.4) and (5.5) are sufficient for the correctness of EDF-VD(k), it remains to show that if (5.3) holds, then (5.4) and (5.5) also hold. Rewriting (5.4) gives

$$\begin{aligned}
& \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} \frac{U_\ell(k)}{x} \le 1 \\
\Leftrightarrow \quad & x \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} U_\ell(k) \le x \\
\Leftrightarrow \quad & \sum_{\ell=k+1}^{K} U_\ell(k) \le x \left(1 - \sum_{\ell=1}^{k} U_\ell(\ell) \right) \\
\Leftrightarrow \quad & \frac{\sum_{\ell=k+1}^{K} U_\ell(k)}{1 - \sum_{\ell=1}^{k} U_\ell(\ell)} \le x, \qquad (5.6)
\end{aligned}$$

where in the last step we used $1 - \sum_{\ell=1}^{k} U_\ell(\ell) > 0$. Rewriting (5.5) gives

$$\begin{aligned} & x \sum_{\ell=1}^{k} U_\ell(\ell) + \sum_{\ell=k+1}^{K} U_\ell(\ell) \leq 1 \\ \Leftrightarrow \quad & x \sum_{\ell=1}^{k} U_\ell(\ell) \leq 1 - \sum_{\ell=k+1}^{K} U_\ell(\ell) \\ \Leftrightarrow \quad & x \leq \frac{1 - \sum_{\ell=k+1}^{K} U_\ell(\ell)}{\sum_{\ell=1}^{k} U_\ell(\ell)}. \end{aligned} \tag{5.7}$$

Hence, if (5.3) holds, there must exist an x that satisfies both (5.6) and (5.7). This concludes the proof of Theorem 12. Note that any x satisfying (5.6) and (5.7) suffices as a scaling parameter. □

5.3.3 Speedup bounds for two and three levels

Between the sufficient condition for schedulability given by (5.3) and the necessary condition (5.1) there is a gap. Note that the necessary conditions in (5.1) would be sufficient if the scheduler was clairvoyant and would know in which level the system was going to be. To analyze this gap due to non-clairvoyance, we consider a task system satisfying the necessary condition given in (5.1) and determine how much faster the processor should be to correctly schedule the task set by EDF-VD.

Recall from Section 1.3.2, that the *speedup factor* of a scheduling algorithm A is the smallest real number σ such that any task system $\mathcal{T}$ that is feasible on a unit-speed processor (in the sense of Definition 4) is correctly scheduled by A on a speed-σ processor. Also recall that the smaller the speedup factor is, the closer the behavior of the algorithm is to that of a clairvoyant exact algorithm.

We provide speedup bounds for EDF-VD when K equals 2 or 3. The computation of speedup bounds when $K > 3$ is an analytically challenging problem that we leave open.

Two levels

Theorem 13. *If a 2-level task system $\mathcal{T}$ satisfies*

$$\max\{U_1(1) + U_2(1), U_2(2)\} \leq 1,$$

then EDF-VD *correctly schedules $\mathcal{T}$ on a processor of speed* 4/3*. In particular, if $\mathcal{T}$ is schedulable on a unit-speed processor, then it is schedulable by* EDF-VD *on a processor of speed* 4/3*.*

Proof. For a 2-level system, the schedulability condition (5.3) is

$$1 - U_1(1) > 0 \quad \text{and} \quad \frac{U_2(1)}{1 - U_1(1)} \leq \frac{1 - U_2(2)}{U_1(1)}. \tag{5.8}$$

To prove the claim, we find the largest q such that, if

$$\begin{aligned} U_1(1) + U_2(1) &\leq q, \\ U_2(2) &\leq q, \end{aligned} \tag{5.9}$$

then sufficient condition (5.8) still holds. The required speedup then equals $1/q$. Note that since $U_2(2)$ only appears with a negative sign on the right-hand side of (5.8) and $U_2(1)$ only appears on the left-hand side, the worst case in terms of tightness of the condition is that $U_2(2) = q$ and $U_2(1) = q - U_1(1)$. Henceforth, this assumption is made. Further, we define $y := 1 - q$. Then, sufficient condition (5.8) becomes

$$\frac{1 - y - U_1(1)}{1 - U_1(1)} \leq \frac{y}{U_1(1)}. \tag{5.10}$$

This condition is satisfied if and only if $U_1(1)^2 - U_1(1) + y \geq 0$ (where we use that $1 - U_1(1) > 0$). This inequality has at most one solution if $y \geq 1/4$. Hence, the largest q such that (5.9) implies (5.8) is $q = 3/4$, and the required speedup is $1/q = 4/3$.

The second part of the statement follows from the first and Proposition 10. □

Three levels

Theorem 14. *If a 3-level task system $\mathcal{T}$ satisfies*

$$\max_{k=1,2,3} \sum_{\ell=k}^{K} U_\ell(k) \leq 1, \tag{5.11}$$

then EDF-VD *correctly schedules $\mathcal{T}$ on a processor of speed 2. In particular, if $\mathcal{T}$ is feasible on a unit-speed processor, then it is schedulable by* EDF-VD *on a processor of speed 2.*

Proof. We recall the schedulability conditions for a 3-level system from (5.3) for $k = 1, 2$:

$$1 - U_1(1) > 0 \quad \text{and} \quad \frac{U_2(1) + U_3(1)}{1 - U_1(1)} \leq \frac{1 - U_2(2) - U_3(3)}{U_1(1)} \tag{5.12}$$

$$1 - U_1(1) - U_2(2) > 0 \quad \text{and} \quad \frac{U_3(2)}{1 - U_1(1) - U_2(2)} \leq \frac{1 - U_3(3)}{U_1(1) + U_2(2)}, \tag{5.13}$$

We find the speedup in a similar way as for a 2-level system. That is, we search for the largest q, such that if

$$\begin{aligned} U_1(1) + U_2(1) + U_3(1) &\leq q, \\ U_2(2) + U_3(2) &\leq q, \\ U_3(3) &\leq q, \end{aligned}$$

then at least one of the conditions in (5.12) and (5.13) holds. Note that the worst-case speedup appears when the above inequalities hold with equality. To see this, we reason as follows.

The term $U_3(3)$ only appears in the right-hand side of (5.12) and (5.13) and the expressions are monotonically decreasing in $U_3(3)$. Hence, the worst case (lowest right-hand side value) is when $U_3(3)$ is largest, i.e., $U_3(3) = q$.

Since $U_3(2)$ appears only in the left-hand side of (5.13) and this side is monotonically increasing in $U_3(2)$, the worst case is if $U_3(2) = q - U_2(2)$.

Since $U_2(1)$ and $U_3(1)$ appear only in the left-hand side of (5.12) and this side is monotonically increasing in $U_2(1) + U_3(1)$, the worst case is if $U_2(1) + U_3(1) = q - U_1(1)$.

Now we substitute the above expressions in the conditions in (5.12) and (5.13) to obtain the following two conditions

$$\frac{q - U_1(1)}{1 - U_1(1)} \leq \frac{1 - q - U_2(2)}{U_1(1)}$$
$$\frac{q - U_2(2)}{1 - U_1(1) - U_2(2)} \leq \frac{1 - q}{U_1(1) + U_2(2)}.$$

In the two equations above we substitute $y = 1 - q$ and after rewriting we find

$$(1 - U_1(1))(U_1(1) + U_2(2)) \leq y$$
$$(1 - U_2(2))(U_1(1) + U_2(2)) \leq y.$$

It is easily verified that the minimum of these two bounds attains its maximum value when $U_1(1) = U_2(2)$. Hence, if we substitute one for the other, we obtain

$$2U_1(1)^2 - 2U_1(1) + y \geq 0.$$

This has at most one solution if $y \geq 1/2$, which gives $q \leq 1/2$; and that gives a speedup of 2.

Again, the second part of the claim follows from the first and Proposition 10. □

5.3.4 Optimality of EDF-VD for two levels

We now show that—at least in the case of 2-level implicit-deadline systems—EDF-VD is optimal with regard to the speedup factor metric.

Theorem 15. *No non-clairvoyant algorithm for scheduling 2-level implicit-deadline mixed-criticality sporadic task systems can have a speedup bound better than* 4/3.

Proof. Consider the example task system $\mathcal{T} = (\tau_1, \tau_2)$, with the following parameters, where ϵ is an arbitrary small positive number.

τ_i	χ_i	p_i	$c_i(1)$	$c_i(2)$
τ_1	1	2	$1 + \epsilon$	$1 + \epsilon$
τ_2	2	4	$1 + \epsilon$	3

This system is feasible according to Definition 4: EDF would meet all deadlines in scenarios of level 1 (since $U_1(1) + U_2(1) < 1$), while only jobs of τ_2 are required to be scheduled in scenarios of level 2 (and $U_2(2) < 1$).

To see that $\mathcal{T}$ cannot be scheduled correctly by an online scheduler, suppose both tasks were to generate jobs simultaneously. It need not be revealed prior to one of the

jobs receiving $1 + \epsilon$ units of execution, whether the level of the scenario is 1 or 2. We consider two cases.
Case 1: The job from τ_1 receives $1 + \epsilon$ units of execution before the job from τ_2 does. In this case, the scenario is revealed to be of level 2. But now there is not enough time remaining for τ_2's job to complete by its deadline at time instant 4.
Case 2: The first job from τ_2 receives $1 + \epsilon$ units of execution before τ_1's job does. In this case, the scenario is revealed to be of level 1, as the job from τ_2 signals that it has completed execution. There is not enough time now remaining for τ_1's job to complete by its deadline at time 2.
We have thus shown that no non-clairvoyant algorithm can correctly schedule $\mathcal{T}$. The theorem follows, based on the observation that $\max\{U_1(1) + U_2(1), U_2(2)\}$ exceeds 3/4 by an arbitrarily small amount. □

5.4 Fixed priorities for implicit-deadline task systems

In this section we consider a sporadic implicit-deadline MC task system, consisting of two criticality levels to be executed on a single machine. The objective is to schedule the tasks according to *fixed priorities* rather than dynamic priorities like in an EDF-based policy. The *rate-monotonic* priority assignment orders the tasks in non-decreasing order of periods and schedules at any point in time the available job with highest priority in this ordering (i.e., the lowest rank in the ordering). We use the following long-standing result for the non-MC setting.

Proposition 11 ([56])**.** *If a feasible priority assignment exists for implicit-deadline task set $\mathcal{T}$, the rate-monotonic priority assignment is feasible for that task set. Furthermore, there exists a feasible priority assignment if the summed utilization over all tasks from $\mathcal{T}$ is not larger than* $\ln 2 \approx 0.693$.

The algorithm we propose in this section is called RM-VP (Rate-Monotonic with Virtual Periods). If a MC task system is deemed schedulable, *virtual periods* $\hat{p}_i$ are computed for all $\tau_i \in \mathcal{T}$. We give a necessary condition for MC-schedulability on a unit-speed machine and derive a speedup bound of $\phi / \ln 2$, where ϕ equals the golden ratio. Finally, we give an example that shows a lower bound of 2 on the speedup that can be attained by any fixed-priority algorithm for a two-level implicit-deadline task system.

5.4.1 Overview of RM-VP

Let $\mathcal{T}$ be the task system to be scheduled, consisting of two criticality levels. Let $U_1(1)$, $U_2(1)$ and $U_2(2)$ be defined as before. Our algorithm RM-VP has a similar approach as EDF-VD. A schedulability condition is given and two cases are distinguished. In one of the cases, the virtual periods are equal to the original periods, whereas in the other case the periods are scaled by a factor λ.

Algorithm RM-VP All tasks are assigned a virtual period $\hat{p}_i$. The tasks are then ordered in non-decreasing order of $\hat{p}_i$. While the system is in level 1, priorities are assigned

according to this ordering. Then, at any point in time the available job with highest priority (of its task) is scheduled for execution. As soon as the system reaches level 2, the level-1 tasks are discarded and priorities are assigned according to the original periods (yielding the same ordering over the level-2 tasks anyway).

The algorithm distinguishes two cases for the scaling of the periods.

1. If $U_2(2) \leq \frac{U_2(1)}{1-U_2(2)/\ln 2}$, $\hat{p}_i = p_i$ for all τ_i. Thus, the priorities are assigned with respect to the original periods.
2. If $U_2(2) > \frac{U_2(1)}{1-U_2(2)/\ln 2}$, $\hat{p}_i = p_i$ for all $\tau_i \in \mathcal{T}$ with $\chi_i = 1$ and $\hat{p}_i = \lambda p_i$ for all $\tau_i \in \mathcal{T}$ with $\chi_i = 2$, where $\lambda = \frac{U_2(1)}{\ln 2 - U_1(1)}$.

5.4.2 Schedulability conditions

Theorem 16. *If task system $\mathcal{T}$ satisfies the following condition*

$$U_1(1) + \min\left\{U_2(2), \frac{U_2(1)}{1-\frac{U_2(2)}{\ln 2}}\right\} \leq \ln 2, \tag{5.14}$$

our algorithm RM-VP *schedules the task system correctly on a unit-speed processor*

Proof. If condition $U_1(1) + U_2(2) \leq \ln 2$ is satisfied, by Proposition 11, it is clear that we can schedule $\mathcal{T}$ using the rate-monotonic scheduling policy, since in both levels we can schedule each job for its own-level execution time.

If condition $U_1(1) + \frac{U_2(1)}{1-U_2(2)/\ln 2} \leq \ln 2$ is satisfied, we first show that in level 1, all jobs will meet their deadlines when using the rate-monotonic schedule based on the virtual periods. The total utilization of the system in level 1 equals

$$\begin{aligned}
\sum_{\tau_i} \frac{c_i(1)}{\hat{p}_i} &= \sum_{\tau_i:\,\chi_i=1} \frac{c_i(1)}{p_i} + \sum_{\tau_i:\,\chi_i=2} \frac{c_i(1)}{\lambda p_i} \\
&= \sum_{\tau_i:\,\chi_i=1} \frac{c_i(1)}{p_i} + \frac{\ln 2 - U_1(1)}{U_2(1)} \sum_{\tau_i:\,\chi_i=2} \frac{c_i(1)}{p_i} \\
&= U_1(1) + \frac{\ln 2 - U_1(1)}{U_2(1)} U_2(1) \\
&= \ln 2,
\end{aligned}$$

which is a sufficient condition for a system to be feasible for scheduling by a rate-monotonic policy (Proposition 11).

Denote by t^* the moment in time where the system reaches level 2. All level-1 tasks are discarded at this point, and we only need to schedule the level-2 tasks to meet their deadlines. For a job of task τ_i that was released before t^* and is still active at time t^*, its virtual period of length λp_i did not pass yet, since in level 1, the task was guaranteed to meet its deadline. Hence, after t^*, there is at least $(1-\lambda)p_i$ of its period left. This reasoning holds for all level-2 tasks that have an active job at time t^*. So we define a

new task system $\mathcal{T}'$, consisting only of level-2 tasks, with periods $(1-\lambda)p_i$ and the same execution time as before. By (5.14), we know that

$$\begin{aligned} & \frac{U_2(1)}{1-\frac{U_2(2)}{\ln 2}} \leq \ln 2 - U_1(1) \\ \Leftrightarrow \quad & \lambda = \frac{U_2(1)}{\ln 2 - U_1(1)} \leq 1 - \frac{U_2(2)}{\ln 2} \\ \Leftrightarrow \quad & 1-\lambda \geq \frac{U_2(2)}{\ln 2}. \end{aligned}$$

The utilization of the new system $\mathcal{T}'$ in level 2 will be

$$\begin{aligned} \sum_{\tau_i:\, \chi_i=2} \frac{c_i(2)}{(1-\lambda)p_i} &\leq \frac{\ln 2}{U_2(2)} \sum_{\tau_i:\, \chi_i=2} \frac{c_i(2)}{p_i} \\ &= \frac{\ln 2}{U_2(2)} U_2(2) = \ln 2, \end{aligned}$$

and the task system $\mathcal{T}'$ can be scheduled by a rate-monotonic scheduling policy (Proposition 11). Since the level-2 jobs that were active at the time of the criticality change can be scheduled, the level-2 jobs that arrive after the criticality change form no problem at all. □

ALGORITHM 6: Rate Monotonic with Virtual Periods (RM-VP)

Input: task system $\mathcal{T} = (\tau_1, \ldots, \tau_n)$ to be scheduled on a unit-speed preemptive processor with fixed priorities

```
if U_1(1) + U_2(2) ≤ ln 2 then
  for i = 1, 2, ..., n do
    p̂_i ← p_i
  end for
else
  if U_1(1) + U_2(1)/(1 − U_2(2)/ln 2) ≤ ln 2 then
    λ ← U_2(1)/(ln 2 − U_1(1))
    for i = 1, 2, ..., n do
      if χ_i = 1 then
        p̂_i ← p_i
      else
        p̂_i ← λ p_i
      end if
    end for
  else
    return unschedulable
  end if
end if
return (schedulable, (p̂_i)_{i=1}^n)
```

5.4.3 Speedup factor

Theorem 17. *A dual-criticality task system $\mathcal{T}$ satisfying*

$$\max\{U_1(1) + U_2(1), U_2(2)\} \leq 1, \tag{5.15}$$

is schedulable by algorithm RM-VP *on a processor of speed $\phi/\ln 2 \approx 2.334$, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ equals the golden ratio.*

Proof. We prove an equivalent statement: if task system $\mathcal{T}$ satisfies

$$\max\{U_1(1) + U_2(1), U_2(2)\} \leq \ln 2/\phi, \tag{5.16}$$

the system is schedulable on a unit-speed machine. We distinguish two cases:
Case 1: $U_2(1) \geq \frac{1}{\phi} U_1(1)$. Then, by condition (5.16), we have

$$\frac{\ln 2}{\phi} \geq U_1(1) + U_2(1) \geq (1 + \frac{1}{\phi}) U_1(1),$$

which gives

$$U_1(1) \leq \frac{\ln 2/\phi}{1 + 1/\phi} = \frac{\ln 2}{\phi^2}.$$

Hence,

$$\begin{aligned} U_1(1) + U_2(2) &\leq \frac{\ln 2}{\phi^2} + \frac{\ln 2}{\phi} \\ &= \ln 2 \left(\frac{1}{\phi^2} + \frac{1}{\phi} \right) = \ln 2, \end{aligned}$$

since $\frac{1}{\phi^2} + \frac{1}{\phi} = \frac{\phi - 1}{\phi} + \frac{1}{\phi} = 1$. Hence, by Theorem 16, RM-VP correctly schedules this system on a unit-speed processor.
Case 2: $U_2(1) \leq \frac{1}{\phi} U_1(1)$. Then we have by (5.16), that

$$\frac{\ln 2}{\phi} \geq U_1(1) + U_2(1) \geq (\phi + 1) U_2(1).$$

Using that $\phi^2 - \phi = \phi(\phi - 1) = \phi \cdot \frac{1}{\phi} = 1$, we find that $\phi + 1 = \phi + (\phi^2 - \phi) = \phi^2$, the above formula gives

$$U_2(1) \leq \frac{\ln 2/\phi}{\phi^2} = \frac{\ln 2}{\phi^3}.$$

Hence, using Theorem 16, it is sufficient to show that

$$\begin{aligned} U_1(1) + \frac{U_2(1)}{1 - U_2(2)/\ln 2} &= U_1(1) + U_2(1)\left(1 + \frac{U_2(2)/\ln 2}{1 - U_2(2)/\ln 2}\right) \\ &\leq U_1(1) + U_2(1)\left(1 + \frac{1/\phi}{1 - 1/\phi}\right) \\ &= (U_1(1) + U_2(1)) + \frac{U_2(1)}{\phi - 1} \\ &\leq \frac{\ln 2}{\phi} + \frac{\ln 2}{\phi^3}\phi \\ &= \ln 2\left(\frac{1}{\phi} + \frac{1}{\phi^2}\right) = \ln 2, \end{aligned}$$

and RM-VP correctly schedules this system on a unit-speed processor. □

5.4.4 Lower bound on speedup

We show that there exist instances with utilization greater than $\frac{1}{2} + \epsilon$ which are not MC-schedulable by any fixed-priority algorithm on a unit-speed processor, for any $\epsilon \in [0, 1]$, but that are MC-schedulable under dynamic priorities.

Let us consider the following task system for $k \in \mathbb{N}$, $c_1 < p_1$, $p_1 > 1$, and $f \in [0, 1]$.

τ_i	χ_i	p_i	$c_i(1)$	$c_i(2)$
τ_1	1	p_1	c_1	c_1
τ_2	2	$kp_1 + 1$	$(f - \frac{c_1}{p_1})p_2$	fp_2

The utilizations in level 1 and 2 are

$$\begin{aligned} U_1(1) + U_2(1) &= \frac{c_1}{p_1} + \frac{c_2(1)}{p_2} = f, \\ U_2(2) &= \frac{c_2(2)}{p_2} = f. \end{aligned}$$

If τ_1 has a higher priority than τ_2 and the system reveals to be in level 2, then, in order to schedule the first job of τ_2, we need that

$$\begin{aligned} & \left\lceil \frac{p_2}{p_1} \right\rceil c_1 + c_2(2) \leq p_2 \\ \Leftrightarrow \quad & (k+1)c_1 + fp_2 \leq p_2 \\ \Leftrightarrow \quad & f \leq 1 - c_1 \frac{k+1}{kp_1 + 1}. \end{aligned} \tag{5.17}$$

If τ_2 has a higher priority than τ_1 and the system reveals to be in level 1, then, in

order to schedule the first job of τ_1, we need that

$$\begin{aligned} & c_2(1) + c_1 \leq p_1 \\ \Leftrightarrow \quad & \left(f - \frac{c_1}{p_1}\right)(kp_1 + 1) + c_1 \leq p_1 \\ \Leftrightarrow \quad & f \leq \frac{p_1 - c_1}{kp_1 + 1} + \frac{c_1}{p_1}. \end{aligned} \tag{5.18}$$

The right-hand side of (5.17) tends to $1 - \frac{c_1}{p_1}$ as k tends to infinity, while the right-hand side of (5.18) tends to $\frac{c_1}{p_1}$ as k tends to infinity. It follows that, for k big enough, if $f > \max\{1 - \frac{c_1}{p_1}, \frac{c_1}{p_1}\}$, then the system is not schedulable by any fixed-priority algorithm. As $1 - \frac{c_1}{p_1}$ is decreasing with c_1 and $\frac{c_1}{p_1}$ is increasing with c_1, we get a the maximum value for f when $1 - \frac{c_1}{p_1} = \frac{c_1}{p_1}$, that is, $c_1 = \frac{1}{2}p_1$ and hence $f = \frac{1}{2}$. It follows that for $f > \frac{1}{2}$, the system is not MC-schedulable by any fixed-priority algorithm with speed 1.

However, the system is MC-schedulable when using EDF-VD. In fact, the schedulability condition for a two-level system (see (5.8)) will become

$$\frac{f - \frac{c_1}{p_1}}{1 - \frac{c_1}{p_1}} \leq \frac{1 - f}{\frac{c_1}{p_1}}$$

giving

$$f \leq 1 - \frac{c_1}{p_1} + \left(\frac{c_1}{p_1}\right)^2. \tag{5.19}$$

Note that $1 - \frac{c_1}{p_1} + \left(\frac{c_1}{p_1}\right)^2 \geq \frac{3}{4}$, for any $\frac{c_1}{p_1} \in [0, 1]$.

Now, as shown above, if $f = \frac{1}{2} + \epsilon$ and $c_1 = \frac{1}{2}p_1$, then inequalities (5.17) and (5.18) are not satisfied while (5.19) is satisfied. Hence, this task system $\mathcal{T}$ is schedulable by EDF-VD, but not by RM-VP.

5.5 Finitely many jobs on multiple processors

In the final section of this chapter, we do not schedule a sporadic task system, but a finite collection of independent jobs. Whereas in the sporadic task model each task could generate infinitely many jobs, here all jobs are known and so are their release times. For a single machine and a 2-level system of independent jobs, Baruah et al. [13] analyzed the *Own Criticality Based Priority* (OCBP) rule and showed that it guarantees a speedup bound of ϕ on a collection of independent jobs (where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ denotes the golden ratio). We show that this approach can be extended to multiple identical machines at the cost of a slightly increased bound.

The OCBP algorithm uses an approach that is commonly referred to in the real-time scheduling literature as the "Audsley approach" [6]. A priority ordering of the jobs is recursively determined off-line, before knowing the actual execution times. The OCBP schedulability test starts by determining the lowest-priority job: job J_h is assigned lowest

priority (priority parameter π_h will be assigned highest rank) if there is at least $c_h(\chi_h)$ time in $[a_h, d_h]$ assuming that every other job J_j is executed before J_h for $c_j(\chi_h)$ units of time. This procedure is recursively applied to the collection of jobs obtained by excluding the jobs that have a priority assigned already, until all the jobs are ordered or no lowest-priority job can be found. A collection of jobs is called *OCBP-schedulable* if the algorithm finds a complete ordering of the jobs. During run time, for each scenario, at each moment in time, the available job with the highest priority is executed.

Theorem 18. *Let $\tilde{J}$ be a collection of jobs that is schedulable on m unit-speed processors. Then $\tilde{J}$ is schedulable using OCBP on m processors of speed $\phi + 1 - 1/m$.*

Proof. We will prove this theorem by contradiction. Let $\tilde{J}$ be a minimal collection of jobs that is MC-schedulable on m unit-speed processors, but not OCBP-schedulable on m processors of speed $\sigma > 1$. By a minimal instance we mean that a proper subset of $\tilde{J}$ is OCBP-schedulable on m speed-σ processors.

Let $\omega_1 = \sum_{j \in \tilde{J}: \chi_j = 1} c_j(1)$ denote the cumulative WCET for jobs with criticality level 1, and let $\omega_2(1) = \sum_{j \in \tilde{J}: \chi_j = 2} c_j(1)$ and $\omega_2(2) = \sum_{j \in \tilde{J}: \chi_j = 2} c_j(2)$ denote the cumulative WCETs for jobs with criticality level 2, at levels 1 and 2, respectively. Let $\tilde{d}_1$ and $\tilde{d}_2$ denote the latest deadlines at levels 1 and 2, respectively and let $\tilde{J}_1$ and $\tilde{J}_2$ denote the corresponding jobs.

Fact 4. *The job in $\tilde{J}$ with latest deadline must be of criticality 2, that is, $\tilde{d}_1 < \tilde{d}_2$.*

Proof. Suppose that $\tilde{d}_1 \geq \tilde{d}_2$. Consider the collection of jobs $\tilde{J}'$ obtained from $\tilde{J}$ by setting the level-2 WCETs of criticality-2 jobs to their level-1 WCET. The MC-schedulability of $\tilde{J}$ implies that also $\tilde{J}'$ is MC-schedulable.

Note that $\tilde{J} \backslash \tilde{J}_1$ should be OCBP-schedulable by the assumed minimality of the instance. If $\tilde{J}_1$ can be assigned the lowest priority, and $\tilde{J} \backslash \tilde{J}_1$ is OCBP-schedulable, then the entire instance is OCBP-schedulable. Since we have assumed this is not the case, $\tilde{J}_1$ can *not* be assigned the lowest priority. But this implies that there is not enough available execution time between the release time of $\tilde{J}_1$ and $\tilde{d}_1$ for executing $\tilde{J}_1$ if all the other jobs

ALGORITHM 7: Own Criticality Based Priority (OCBP)

Input: job set $J = (J_1, \ldots, J_n)$ to be scheduled on m preemptive processors

for $k = 1$ to n **do**
  if $\exists\, J_h \in J$ such that there is at least $c_h(\chi_h)$ time in $[a_h, d_h]$ assuming every other job J_j is executed before J_h for $c_j(\chi_h)$ units of time **then**
    priority $\pi_h \leftarrow n - k + 1$ (where lower index denotes higher priority)
    $J \leftarrow J \setminus \{J_h\}$
  else
    return not OCBP-schedulable
  end if
end for
return (OCBP-schedulable, $(\pi_h)_{h=1}^n$)

J_j are executed for $c_j(1)$ units of time. Then, level-1 behaviors of $\tilde{J}'$ cannot be scheduled, which is a contradiction to the MC-schedulability of $\tilde{J}'$. Hence, $\tilde{d}_1 < \tilde{d}_2$. □

A *work-conserving schedule* is a schedule that never leaves a processor idle if there is a job available. We define Λ_1 as the set of time intervals in $[0, \tilde{d}_1)$ on which all processors are idle, for any work-conserving schedule. For the interval $[\tilde{d}_1, \tilde{d}_2)$, Λ_2 is defined similarly. For $\ell \in \{1, 2\}$, λ_ℓ is defined as the total length of the intervals in Λ_ℓ.

Fact 5. *For $\ell \in \{1, 2\}$, and for each job J_j in $\tilde{J}$ such that $\chi_j \leq \ell$, we have $[a_j, d_j] \cap \Lambda_\ell = \emptyset$. This implies that $\lambda_2 = 0$.*

Proof. Any job J_j in $\tilde{J}$ such that $\chi_j \leq \ell$ and $[a_j, d_j] \cap \Lambda_\ell \neq \emptyset$ would meet its deadline if it were assigned lowest priority. As $\tilde{J}$ is not OCBP-schedulable on speed-σ processors, the collection of jobs obtained by removing such J_j from $\tilde{J}$ is also not OCBP-schedulable. This contradicts the assumed minimality of $\tilde{J}$. □

Since $\tilde{J}$ is MC-schedulable on m speed-1 processors, ω_1 cannot exceed $m(\tilde{d}_1 - \lambda_1)$ in any criticality-1 scenario. Moreover, in scenarios where all jobs execute for their WCET at criticality 1, $\omega_1 + \omega_2(1)$ cannot exceed $m(\tilde{d}_2 - \lambda_1)$ and in scenarios where all jobs execute for their WCET at criticality 2, $\omega_2(2)$ cannot exceed $m(\tilde{d}_2 - \lambda_2)$. Hence, the following inequalities hold:

$$\omega_1 \leq m(\tilde{d}_1 - \lambda_1), \tag{5.20}$$

$$\omega_1 + \omega_2(1) \leq m(\tilde{d}_2 - \lambda_1) \leq m\tilde{d}_2, \tag{5.21}$$

$$\omega_2(2) \leq m(\tilde{d}_2 - \lambda_2) = m\tilde{d}_2. \tag{5.22}$$

Since $\tilde{J}$ is not OCBP-schedulable on m speed-σ processor and it is a minimal instance, $\tilde{J}_1$ and $\tilde{J}_2$ cannot be the lowest-priority jobs. This implies that

$$\frac{1}{m}(\omega_1 + \omega_2(1) - c_{\tilde{J}_1}(1)) + c_{\tilde{J}_1}(1) > \sigma(\tilde{d}_1 - \lambda_1),$$
$$\frac{1}{m}(\omega_1 + \omega_2(2) - c_{\tilde{J}_2}(2)) + c_{\tilde{J}_2}(2) > \sigma(\tilde{d}_2 - \lambda_2) = \sigma\tilde{d}_2.$$

Hence,

$$\omega_1 + \omega_2(1) > \sigma m(\tilde{d}_1 - \lambda_1) - (m-1)c_{\tilde{J}_1}(1), \tag{5.23}$$

$$\omega_1 + \omega_2(2) > \sigma m\tilde{d}_2 - (m-1)c_{\tilde{J}_2}(2). \tag{5.24}$$

By inequalities (5.21) and (5.23), it follows that

$$m\tilde{d}_2 > \sigma m(\tilde{d}_1 - \lambda_1) - (m-1)c_{\tilde{J}_1}(1)$$
$$\Leftrightarrow \quad \left(1 - \frac{1}{m}\right) c_{\tilde{J}_1}(1) + \tilde{d}_2 > \sigma(\tilde{d}_1 - \lambda_1).$$

Let us define $\psi = (\tilde{d}_1 - \lambda_1)/\tilde{d}_2$. By MC-schedulability, we have: $c_{\tilde{J}_1}(1) \leq \tilde{d}_1 - \lambda_1$. Therefore,

$$\left(1 - \frac{1}{m}\right)(\tilde{d}_1 - \lambda_1) + \tilde{d}_2 > \sigma(\tilde{d}_1 - \lambda_1)$$

$$\Leftrightarrow \quad \sigma < 1 - \frac{1}{m} + \frac{\tilde{d}_2}{\tilde{d}_1 - \lambda_1}$$

$$= 1 + \frac{1}{\psi} - \frac{1}{m}.$$

By inequalities (5.20), (5.22) and (5.24), we obtain

$$m(\tilde{d}_1 - \lambda_1) + m\tilde{d}_2 > \sigma m\tilde{d}_2 - (m-1)c_{\tilde{J}_2}(2)$$

$$\Leftrightarrow \quad \tilde{d}_1 - \lambda_1 + \tilde{d}_2 + (1 - 1/m)c_{\tilde{J}_2}(2) > \sigma\tilde{d}_2.$$

By MC-schedulability, we have $c_{\tilde{J}_2}(2) \leq \tilde{d}_2$. Therefore,

$$\tilde{d}_1 - \lambda_1 + \left(2 - \frac{1}{m}\right)\tilde{d}_2 > \sigma\tilde{d}_2$$

$$\Leftrightarrow \quad \sigma < \frac{\tilde{d}_1 - \lambda_1}{\tilde{d}_2} + 2 - \frac{1}{m}$$

$$= \psi + 2 - \frac{1}{m}.$$

Hence,

$$\sigma < \min\left\{1 + \frac{1}{\psi} - \frac{1}{m}, \psi + 2 - \frac{1}{m}\right\}.$$

As $\psi + 2 - \frac{1}{m}$ increases and $1 + \frac{1}{\psi} - \frac{1}{m}$ decreases with increasing ψ, the minimum value of σ occurs when $\psi + 2 - \frac{1}{m} = 1 + \frac{1}{\psi} - \frac{1}{m}$, that is, when $\psi = \frac{\sqrt{5}-1}{2} = \phi - 1$. Hence, $\sigma < \phi + 1 - \frac{1}{m}$.

Since the assumption that $\tilde{J}$ is not OCBP-schedulable led to the conclusion that $\sigma < \phi + 1 - \frac{1}{m}$, a set of jobs that is MC-schedulable on m unit-speed processors, can be scheduled by OCBP on m processors running at speed $\phi + 1 - \frac{1}{m}$. □

5.6 Epilogue

In this chapter various scheduling problems are considered in the mixed-criticality setting. Many problems that are considered, have natural extensions that remain open. Take for example the result from Section 5.5 concerning a job set on multiple machines. The OCBP scheduling policy works only for 2 criticality levels. A natural extension would be to have a policy for K levels.

For the scheduling policy EDF-VD given in this chapter, we have used a single scaling factor x for tasks from all levels. It could be interesting to see if better speedup bounds

can be attained by choosing different scaling factors for tasks from different criticality levels. Next to that, for $K > 3$, no explicit speedup bound was found at all, which also leaves a direction for further research.

For the scheduling of MC task systems, essentially the only known results are for a single machine. In [15] a small result is given for scheduling an implicit-deadline MC task system on multiple identical machines. There, the utilization vectors of tasks are scheduled to the machines using the vector scheduling approach of Chekuri and Khanna [27] in a black-box manner (see Chapter 4, where we also refer to the Vector Scheduling problem).

A nice extension of different parts of this dissertation would be to consider mixed-criticality scheduling (of implicit-deadline task systems, to start with) on *unrelated* machines.

Bibliography

[1] F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Sviridenko. Approximation schemes for minimizing average weighted completion time with release dates. In *Proceedings of 40th Symposium on Foundations of Computer Science*, pages 32–43. IEEE, 1999.

[2] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of 16th Euromicro Conference on Real-Time Systems*, pages 187–195. IEEE, 2004.

[3] S. Anand, N. Garg, and N. Megow. Meeting deadlines: How much speed suffices? In L. Aceto, M. Henzinger, and J. Sgall, editors, *Proceedings of 38th International Colloquium on Automata, Languages and Programming*, volume 6755 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 2011.

[4] B. Andersson and E. Tovar. Competitive analysis of partitioned scheduling on uniform multiprocessors. In *Proceedings of 21st International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[5] A. Asadpour, U. Feige, and A. Saberi. Santa Claus meets hypergraph matchings. In A. Goel, K. Jansen, J.D.P. Rolim, and R. Rubinfeld, editors, *Approximation, Randomization and Combinatorial Optimization: Algorithms and Techniques*, volume 5171 of *Lecture Notes in Computer Science*, pages 10–20. Springer, 2008.

[6] N.C. Audsley. *Flexible scheduling of hard real-time systems.* PhD thesis, Department of Computer Science, University of York, 1993.

[7] B. Awerbuch, Y. Azar, E.F. Grove, M.-Y. Kao, P. Krishnan, and J.S. Vitter. Load balancing in the L_p norm. In *Proceedings of 36th Symposium on Foundations of Computer Science*, pages 383–391. IEEE, 1995.

[8] Y. Azar and A. Epstein. Convex programming for scheduling unrelated machines. In *Proceedings of 37th Symposium on Theory of Computing*, pages 331–337. ACM, 2005.

[9] N. Bansal, C. Rutten, S. van der Ster, T. Vredeveld, and R. van der Zwaan. Approximating real-time scheduling on identical machines. In A. Pardo and A. Viola, editors, *Proceedings of 11th Latin American Theoretical Informatics Symposium*, volume 8392 of *Lecture Notes in Computer Science*, pages 550–561. Springer, 2014.

[10] N. Bansal and M. Sviridenko. The Santa Claus problem. In *Proceedings of 38th Symposium on Theory of Computing*, pages 31–40. ACM, 2006.

[11] N. Bansal, T. Vredeveld, and R. van der Zwaan. Approximating vector scheduling: Almost matching upper and lower bounds. In A. Pardo and A. Viola, editors, *Proceedings of 11th Latin American Theoretical Informatics Symposium*, volume 8392 of *Lecture Notes in Computer Science*, pages 47–59. Springer, 2014.

[12] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Scoredos, P. Stanfill, D. Stuart, and R. Urzi. White paper: A research agenda for mixed-criticality systems. Technical report, 2009. Available at `http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/`.

[13] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2012.

[14] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proceedings of 24th Euromicro Conference on Real-Time Systems*, pages 145–154. IEEE, 2012.

[15] S. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In C. Demetrescu and M.M. Halldórsson, editors, *Proceedings of 19th European Symposium on Algorithms*, volume 6942 of *Lecture Notes in Computer Science*, pages 555–566. Springer, 2011.

[16] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Systems*, 46(1):3–24, 2010.

[17] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of 26th Real-Time Systems Symposium*, pages 321–329. IEEE, 2005.

[18] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 55(7):918–923, 2006.

[19] S. Baruah and J. Goossens. Scheduling real-time tasks: Algorithms and complexity. In J.Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models and Performance Evaluation*, chapter 28. CRC Press, 2004.

[20] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of 16th Real-Time and Embedded Technology and Applications Symposium*, pages 13–22. IEEE, 2010.

[21] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of 11th Real-Time Systems Symposium*, pages 182–190. IEEE, 1990.

[22] D. Bertsimas and J.N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, 1997.

[23] V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. A constant-approximate feasibility test for multiprocessor real-time scheduling. *Algorithmica*, 62(3–4):1034–1049, 2012.

[24] J. Bruno, E.G. Coffman Jr., and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17(7):382–387, 1974.

[25] A. Burns and R. Davis. Mixed criticality systems - A review. Technical report, 2013. Available at `http://www-users.cs.york.ac.uk/~burns/review.pdf`.

[26] S. Chakraborty, S. Künzli, and L. Thiele. Approximate schedulability analysis. In *Proceedings of 23rd Real-Time Systems Symposium*, pages 159–168. IEEE, 2002.

[27] C. Chekuri and S. Khanna. On multi-dimensional packing problems. *SIAM Journal on Computing*, 33(4):837–851, 2004.

[28] B. Chen, Y. Ye, and J. Zhang. Lot-sizing scheduling with batch setup times. *Journal of Scheduling*, 9(3):299–310, 2006.

[29] J.-J. Chen and S. Chakraborty. Resource augmentation bounds for approximate demand bound functions. In *Proceedings of 32nd Real-Time Systems Symposium*, pages 272–281. IEEE, 2011.

[30] J.-J. Chen and S. Chakraborty. Partitioned packing and scheduling for sporadic real-time tasks in identical multiprocessor systems. In *Proceedings of 24th Euromicro Conference on Real-Time Systems*, pages 24–33. IEEE, 2012.

[31] J. Correa, A. Marchetti-Spaccamela, J. Matuschke, O. Svensson, L. Stougie, V. Verdugo, and J. Verschae. Strong LP formulations for scheduling splittable jobs on unrelated machines. In *Proceedings of 17th Conference on Integer Programming and Combinatorial Optimization*, to appear, 2014.

[32] M. Drozdowski. *Scheduling for Parallel Processing*. Springer, 2009.

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

[34] A. Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *Proceedings of 34th Real-Time Systems Symposium*, pages 78–87. IEEE, 2013.

[35] T. Ebenlendr, M. Krčál, and J. Sgall. Graph balancing: A special case of scheduling unrelated parallel machines. In *Proceedings of 19th Symposium on Discrete Algorithms*, pages 483–490. SIAM-ACM, 2008.

[36] F. Eisenbrand and T. Rothvoß. A PTAS for static priority real-time scheduling with resource augmentation. In L. Aceto, I. Damgård, L.A. Goldberg, M.M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Proceedings of 35th International Colloquium on Automata, Languages and Programming*, volume 5125 of *Lecture Notes in Computer Science*, pages 246–257. Springer, 2008.

[37] F. Eisenbrand and T. Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *Proceedings of 21st Symposium on Discrete Algorithms*, pages 1029–1034. SIAM-ACM, 2010.

[38] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proceedings of 24th Euromicro Conference on Real-Time Systems*, pages 135–144. IEEE, 2012.

[39] P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-Time Systems*, 50(1):48–86, 2014.

[40] N. Fisher, S. Baruah, and T.P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Proceedings of 18th Euromicro Conference on Real-Time Systems*, pages 118–127. IEEE, 2006.

[41] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[42] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5(2):287–326, 1979.

[43] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *Proceedings of 32nd Real-Time Systems Symposium*, pages 13–23. IEEE, 2011.

[44] K. Jain. A factor 2 approximation algorithm for the generalized Steiner network problem. *Combinatorica*, 21(1):39–60, 2001.

[45] K. Jansen and L. Porkolab. Improved approximation schemes for scheduling unrelated parallel machines. In *Proceedings of 31st Symposium on Theory of Computing*, pages 408–417. ACM, 1999.

[46] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000.

[47] H. Karloff. *Linear Programming*. Birkhäuser, 1991.

[48] R.M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 40:85–103, 1972.

[49] R.M. Karp, F.T. Leighton, R.L. Rivest, C.D Thompson, U.V. Vazirani, and V.V. Vazirani. Global wire routing in two-dimensional arrays. *Algorithmica*, 2:113–129, 1987.

[50] V.S.A. Kumar, M.V. Marathe, S. Parthasarathy, and A. Srinivasan. Approximation algorithms for scheduling on multiple machines. In *Proceedings of 46th Symposium on Foundations of Computer Science*, pages 254–263. IEEE, 2005.

[51] L.C. Lau, R. Ravi, and M. Singh. *Iterative methods in combinatorial optimization.* Cambridge University Press, 2011.

[52] J.K. Lenstra. The mystical power of twoness: in memoriam Eugene L. Lawler. *Journal of Scheduling*, 1(1):3–14, 1998.

[53] J.K. Lenstra, D.B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1–3):259–271, 1990.

[54] J.Y.-T. Leung. Some basic scheduling algorithms. In J.Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models and Performance Evaluation*, chapter 3. CRC Press, 2004.

[55] H. Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proceedings of 16th Real-Time Systems Symposium*, pages 183–192. IEEE, 2010.

[56] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[57] Z. Liu and T.C.E. Cheng. Minimizing total completion time subject to job release dates and preemption penalties. *Journal of Scheduling*, 7:313–327, 2004.

[58] A. Marchetti-Spaccamela, C. Rutten, S. van der Ster, and A. Wiese. Assigning sporadic tasks to unrelated parallel machines. In A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, editors, *Proceedings of 39th International Colloquium on Automata, Languages and Programming*, volume 7391 of *Lecture Notes in Computer Science*, pages 665–676. Springer, 2012.

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

[60] C.N. Potts and L.N. van Wassenhove. Integrating scheduling with batching and lot-sizing: A review of algorithms and complexity. *Journal of the Operational Research Society*, 43(5):395–406, 1992.

[61] G. Raravi, B. Andersson, and K. Bletsas. Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. *Real-Time Systems*, 49(1):29–72, 2013.

[62] G. Raravi and V. Nélis. A PTAS for assigning sporadic tasks on two-type heterogeneous multiprocessors. In *Proceedings of 33rd Real-Time Systems Symposium*, pages 117–126. IEEE, 2012.

[63] F. Schalekamp, R. Sitters, S. van der Ster, L. Stougie, V. Verdugo, and A. van Zuylen. Split scheduling with uniform setup times. *Journal of Scheduling*, to appear. Available at `http://link.springer.com/article/10.1007%2Fs10951-014-0370-4`.

[64] P. Schuurman and G.J. Woeginger. Preemptive scheduling with job-dependent setup times. In *Proceedings of 10th Symposium on Discrete Algorithms*, pages 759–767. SIAM-ACM, 1999.

[65] D.B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1–3):461–474, 1993.

[66] M. Singh and L.C. Lau. Approximating minimum bounded degree spanning trees to within one of optimal. In *Proceedings of 39th Symposium on Theory of Computing*, pages 661–670. ACM, 2007.

[67] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[68] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proceedings of Conference on Design, Automation & Test in Europe*, pages 147–152, 2013.

[69] O. Svensson. Santa Claus schedules jobs on unrelated machines. *SIAM Journal on Computing*, 41(5):1318–1341, 2012.

[70] S. van der Ster. The allocation of scarce resources in disaster relief. Master's thesis, VU University, Amsterdam, 2010.

[71] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of 28th Real-Time Systems Symposium*, pages 239–243. IEEE, 2007.

[72] A. Wiese, V. Bonifaci, and S. Baruah. Partitioned EDF scheduling on a few types of unrelated multiprocessors. *Real-Time Systems*, 49(2):219–238, 2013.

[73] W. Xing and J. Zhang. Parallel machine scheduling with splitting jobs. *Discrete Applied Mathematics*, 103(1–3):259–269, 2000.

Samenvatting

Het benaderen van uitvoerbaarheid in real-time scheduling

De snelheid opvoeren om deadlines te halen

Het vakgebied van *scheduling* houdt zich bezig met het toewijzen van middelen aan een verzameling van opdrachten die moeten worden uitgevoerd. Problemen die in dit vakgebied naar voren komen hebben met elkaar gemeen dat er wordt gezocht naar de *beste* manier om die middelen toe te wijzen aan de opdrachten. Wat "de beste" manier is kan variëren tussen de verschillende problemen en ook andere probleemparameters kunnen verschillende vormen aannemen.

De middelen waarmee deze taken worden uitgevoerd worden *machines* genoemd. Daarbij kan gedacht worden aan een machine in een fabriek die bijvoorbeeld losse onderdelen tot een eindproduct verwerkt, maar de machines in het model zouden in werkelijkheid ook teams van mensen kunnen voorstellen die bepaalde opdrachten moeten volbrengen. De oplossing die we vinden voor een gegeven probleem noemen we een *schedule* (schema). Het vinden van het optimale schedule is in veel gevallen NP-moeilijk, waardoor we onze toevlucht moeten nemen tot *benaderingsalgoritmen*.

In Hoofdstuk 1 leggen we eerst nauwkeuriger uit wat schedulingproblemen precies zijn en wat het verschil is tussen klassieke schedulingproblemen en problemen in de real-time scheduling. Vervolgens behandelen we kort wat complexiteitstheorie en introduceren we benaderingsalgoritmen.

Het probleem dat in Hoofdstuk 2 wordt bekeken is een zogeheten klassiek schedulingprobleem. Daarin hebben we te maken met een verzameling opdrachten die moet worden ingedeeld op meerdere machines. Voor iedere opdracht kijken we naar de *completeringstijd* van de opdracht, dat wil zeggen, het moment dat de opdracht voltooid wordt. We pogen de som van deze completeringstijden te minimaliseren. Hierbij mogen de opdrachten worden opgesplitst in meerdere delen (die ook tegelijkertijd op meerdere machines mogen worden uitgevoerd), maar voordat een (deel van een) opdracht kan worden uitgevoerd is er een instellingstijd s nodig. Tijdens het instellen kan een machine niet gebruikt worden voor het uitvoeren van een opdracht, of voor het instellen van een andere opdracht. De instellingstijd s wordt verondersteld onafhankelijk te zijn van de opdracht, de machine waarop de opdracht wordt uitgevoerd, en de volgorde van de opdrachten. Voor het speciale geval van twee machines geven we een algoritme dat in polynomiale tijd een optimaal schedule vindt. Voor het geval van meerdere machines geven we aan waarom de oplossing voor twee machines niet eenvoudig te generaliseren valt. Wel geven we een simpel benaderingsalgoritme met een benaderingsgarantie van $2 + \frac{1}{4}(\sqrt{17} - 1) \approx 2.781$ voor het

algemene geval.

De overige drie hoofdstukken in dit proefschrift behandelen problemen uit de *real-time scheduling*. In real-time scheduling gaat het erom te bepalen wat de uitvoerbaarheid is van een *systeem van taken* op een verzameling machines. Een *taak* geeft herhaaldelijk een opdracht af die moet worden voltooid voor een gegeven deadline. De frequentie waarmee deze opdrachten worden afgegeven is bekend en iedere opdracht van dezelfde taak heeft dezelfde uitvoeringstijd en dezelfde relatieve deadline. Een takensysteem bestaat uit meerdere taken, ieder met zijn eigen parameters. Een *uitvoerbaarheidstest* moet uitwijzen of een gegeven systeem van taken kan worden uitgevoerd op een gegeven verzameling machines, zodanig dat iedere opdracht zijn deadline haalt.

In het probleem beschreven in Hoofstuk 3 zijn de machines *ongerelateerd*. Dit betekent dat de uitvoeringstijd van een opdracht afhangt van de machine waarop de opdracht wordt uitgevoerd. We zoeken naar een splitsing van de taken over de machines zodat alle opdrachten van een taak worden uitgevoerd op de machine waaraan de taak is toegewezen, zodanig dat alle opdrachten hun deadlines halen. Voor het algemene geval geven we een benaderingsalgoritme dat de taken zo verdeelt over de machines dat alle deadlines worden gehaald als de machines een factor $8 + 2\sqrt{6} \approx 12.9$ sneller werken, gegeven dat het systeem uitvoerbaar is op machines zonder extra snelheid. Om dit resultaat te behalen introduceren we lineaire benaderingen van de *vraagbegrenzingsfunctie* (demand bound function) en daarbovenop ontwikkelen we een nieuw afrondingsalgoritme voor lineaire programma's (LP's). Voor het geval dat het aantal machines een constante is, geven we een Polynomiale Tijd Approximatie Schema (PTAS). Voor een kleine constante $\epsilon > 0$, bepaalt deze test of een systeem van taken kan worden ingedeeld op de machines als deze een snelheid $1+O(\epsilon)$ krijgen, of dat het systeem niet uitvoerbaar is als de machines snelheid 1 hebben. De looptijd van deze test is polynomiaal in de grootte van de probleeminstantie, maar kan exponentieel zijn in $1/\epsilon$.

In Hoofdstuk 4 beschouwen we identieke machines en bouwen voort op het PTAS uit het voorgaande hoofdstuk. We vinden een $(1 + \epsilon)$-benaderingsalgoritme voor het probleem waarin we een verzameling taken willen opsplitsen over een willekeurig aantal identieke machines. Hiermee verbeteren we ten opzichte van resultaat van anderen in de looptijd van het algoritme. Er was een PTAS bekend voor het geval dat de verhouding tussen de maximum en minimum relatieve deadline een constante is. Ons algoritme geeft een exponentiële verbetering in de looptijd en kan daardoor een breder scala aan probleeminstanties verwerken. Als bijvoorbeeld de verhouding tussen de maximum en minimum relatieve deadline polynomiaal in het aantal taken is, heeft ons algoritme een quasi-polynomiale looptijd, terwijl het vorige resultaat al op een exponentiële looptijd uitkwam.

Het laatste hoofdstuk behandelt scheduling in de *mixed-criticality* omgeving. Dit houdt in dat iedere taak een gegeven *criticality niveau* heeft, dat aangeeft in hoeverre een taak kritiek is. Het systeem bevindt zich ook in één van deze K niveaus en elke taak heeft verschillende uitvoeringstijden voor de verschillende niveaus waarin het systeem zich kan bevinden. We bestuderen de uitvoerbaarheid van een mixed-criticality takensysteem met impliciete deadlines op één processor. Hiervoor geven we een schedulingalgoritme EDF-VD dat in polynomiale tijd beslist of een takensysteem van K niveaus kan worden uitgevoerd op een snellere processor, of dat het niet uitvoerbaar is als de processor snelheid

1 heeft. De extra benodigde snelheid hangt af van het aantal niveaus. Voor een systeem met 2 niveaus laten we zien dat een snelheid van ten hoogste 4/3 benodigd is, terwijl voor een systeem van 3 niveaus deze snelheid ten hoogste 2 is. Verder tonen we aan dat geen enkel *niet-helderziend* algoritme (dat niet van te voren weet in welk niveau het systeem zich zal bevinden) de uitvoerbaarheid van een takensysteem correct kan bepalen met een snelheid minder dan 4/3.

Voor systemen bestaande uit twee niveaus geven we ook een algoritme dat werkt met *vaste prioriteiten*. Dit houdt in dat vooraf een vaste ordening van de taken wordt gemaakt en de opdrachten die door deze taken worden afgegeven worden afgehandeld volgens de prioriteiten bepaald door deze ordening. Ons vaste-prioriteitenalgoritme RM-VP bepaalt of een takensysteem kan worden uitgevoerd op één processor die snelheid $\phi / \ln 2 \approx 2.334$ heeft, of dat het niet kan worden uitgevoerd op een processor met snelheid 1, waarbij $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ de gulden snede is. Ten slotte geven we een vaste-prioriteitenalgoritme dat we OCBP noemen voor het uitvoeren van een verzameling van mixed-criticality opdrachten op m identieke machines. Dit algoritme bepaalt of de opdrachten kunnen worden uitgevoerd zodat alle deadlines worden gehaald als de machines snelheid $\phi + 1 - \frac{1}{m}$ hebben, of dat ze niet kunnen worden uitgevoerd als de machines snelheid 1 hebben.

Acknowledgments

This dissertation would never have seen daylight if it was not for my promotor Leen Stougie. After supervising my Master's thesis he asked me apply for the PhD project that he filed with the faculty and that is where my adventure began. I wish to thank Leen for his support and guidance during the past years. I am glad that we got to work together on some nice projects. Apart from working together in Amsterdam, I enjoyed the work trips that we made, to Rome, Berlin, Corsica, and other places, and the chats we had about doing sports and life in general. I am grateful that Leen introduced me to lots of people in the OR community, from which I benefitted a lot.

I am also indebted to my other promotor Alberto Marchetti-Spaccamela. Alberto is one of the people I was introduced to through Leen and I am happy that he took up the role as my promotor next to Leen. Thank you, Alberto, for making it possible for me to visit La Sapienza for two extended periods, adding up to six months in total. In terms of productivity, especially the first visit was extremely fruitful and led to a wonderful paper.

Next, I would like to thank the thesis committee, consisting of Sanjoy Baruah, Vincenzo Bonifaci, Nicole Megow and René Sitters for their assessment of this dissertation and their helpful comments to improve it.

Then, there is a long list of co-authors that I would like to thank for their collaboration. The first project I got involved in after starting the PhD trajectory was with René Sitters, Sylvia Boyd and Leen. Thank you for letting me join your project at a later stage. Although the paper that this project led to did not end up in this dissertation, it got me my first publication and allowed me to already get a feeling for writing papers in an early stage of my PhD.

The initial topic of my PhD trajectory was "Mixed-criticality scheduling" and the research in this area led to the findings in Chapter 5. I want to thank my co-authors Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela and Leen Stougie for collaboration on this theme.

My first longer visit to Rome was in the fall of 2011. I was lucky that Andreas Wiese and Cyriel Rutten happened to be there during that period and we had a fruitful collaboration together with Alberto that led to the results as presented in Chapter 3. Andreas and Cyriel, thank you for working together, but also for the many (pasta) lunches and all the caffè that we had together.

Following up on some of the work in Rome was a project that in my e-mail inbox was called "Team Eindhoven". Thanks to Cyriel, I was invited to join this project of which the results appear in Chapter 4. This chapter is technically the most challenging from the whole dissertation and it was not so easy to target the right conference for it. I want to thank my co-authors on this paper, Nikhil Bansal, Cyriel Rutten, Tjark Vredeveld and

Ruben van der Zwaan for the efforts needed to obtain the results that we did and to get it published in a suitable conference.

Finally, there is the chapter on split scheduling, for which the inspiration came out of my Master's thesis. This chapter is the one closest to my heart and I wish to thank my co-authors Frans Schalekamp, René Sitters, Leen Stougie, Víctor Verdugo and Anke van Zuylen for their collaboration. The paper would not have the good shape that it does now without their contributions.

During both stays in Rome, Vincenzo Bonifaci was always available for questions, whether it concerned research, practical issues or Italian bureaucratics. Further, Vincenzo, and the following people at La Sapienza made me enjoy the Roman life more: Aris Anagnostopoulos, Marek Adamczyk, Łukasz Jeż and Stefano Leonardi. Thank you for going for caffè with me, for having an *aperitivo*, for taking me for pastries at Regoli, for lunch, and for *gelati* on Wednesdays at the "Palazzo del Freddo".

My second stay in Rome would also not have been the same without the great friends that I met through my Italian language course. Katie Shields, you are a linguistic genius and a wonderful, cheerful person to hang out with. Thank you for the sight seeing and dinners in Rome and for spending a nice (but rainy) weekend in Firenze with me. Marisella Cruz, although our lives are completely different, we found some funny similarities and I always enjoyed hanging out with you! Thank you for inviting me to the beautiful Monestevole, where I hope to be again. Finally, Ed Slesak and Elena Past, thank you for being great friends, for having dinner, discovering new *gelaterias*, discussing politics, Dutch names in Michigan, and anything else you could think of. Elena inspired me to pursue a challenging goal in sports during the writing of my dissertation and this May I will participate in a 100k rowing regatta.

The past four years I have had a pleasant working environment at the Vrije Universiteit. I would like to thank my colleagues at the department of Econometrics and Operations Research for providing that environment. A special thanks for all my office mates over the past four years: Bahar, Chris, Nigel, Warren, Ping, Yuyu, Chao, Joost, Martijn and Max.

My professional life would not have been any success if my personal life was not filled with great friends. Mariëtte, we have been friends from the first day of our Bachelor's in Econometrics and Operations Research. Thank you for being there for me, in good times and not so good. I was very happy when you were accepted at the VU, first for the JR position and then as a PhD. It was great to share the PhD experience with you for the last few years.

Ay May, thank you for designing the wonderful cover of my dissertation. Without you, the cover would have been plain and boring, and you made it much more fun. And that also holds for you being a friend in my life; you make it much more fun!

Thomas, when I persuaded you to come to the "open day" of the Econometrics study in 2003, I guess that neither of us expected that you would be standing by my side as my friend and my paranymph 11 years later. Max, in the introduction week of our studies you did not participate so much and I was afraid that you were a bit boring. I could not have been more wrong! You have become my friend, my party buddy and now my paranymph, too. Guys, thank you for willing to be my paranymphs at my defense. I am sure that I will feel safe, accompanied by my two tall defenders.

Binte and Ramsy, together with Jet, Ay, Thomas en Max, you were my buddies during the Bachelor's phase. I believe that we have all brought each another to better achievements and I want to thank you all for that unique friendship.

I always liked doing activities on the side and the place for me to do that was my rowing club Skøll. Both while studying and during my PhD I have spent a lot of time there and made a lot of friends. My first rowing team FAME'r, fellow board members from Bestuur'08 (lieverds!), my "Koninklijke Clubacht" rowing team, my former roomies, all the lightweight rowers I have coached, my fellow coaches, and everyone else; thank you for being friends and adding so much pleasure to my life.

I wish to thank my parents, Peter and Ineke, for always supporting me, both during the period I was studying (also financially) and during the PhD trajectory. You always let me make my own decisions, but believed in me every step of the way. My beloved sisters, Tamara and Jessica, I am lucky to have such great supportive sisters. It feels great to have you so close to me in Amsterdam and to know that you are always there for me. Pap, mam, Tam and Jes, I love you!

Finally, I am thanking my love Adriaan for his faith in me. He believes in me more than I do, and always encourages me to pursue my dreams. Adriaan, thank you so much for supporting me in writing my dissertation, for kicking my ass when I needed it, and thank you for loving me.

Suzanne van der Ster
Amsterdam, April 2014

The Tinbergen Institute is the Institute for Economic Research, which was founded in 1987 by the Faculties of Economics and Econometrics of the Erasmus University Rotterdam, University of Amsterdam and VU University Amsterdam. The Institute is named after the late Professor Jan Tinbergen, Dutch Nobel Prize laureate in economics in 1969. The Tinbergen Institute is located in Amsterdam and Rotterdam. The following books recently appeared in the Tinbergen Institute Research Series:

536 B. VOOGT, *Essays on Consumer Search and Dynamic Committees*

537 T. DE HAAN, *Strategic Communication: Theory and Experiment*

538 T. BUSER, *Essays in Behavioural Economics*

539 J.A. ROSERO MONCAYO, *On the importance of families and public policies for child development outcomes*

540 E. ERDOGAN CIFTCI, *Health Perceptions and Labor Force Participation of Older Workers*

541 T. WANG, *Essays on Empirical Market Microstructure*

542 T. BAO, *Experiments on Heterogeneous Expectations and Switching Behavior*

543 S.D. LANSDORP, *On Risks and Opportunities in Financial Markets*

544 N. MOES, *Cooperative decision making in river water allocation problems*

545 P. STAKENAS, *Fractional integration and cointegration in financial time series*

546 M. SCHARTH, *Essays on Monte Carlo Methods for State Space Models*

547 J. ZENHORST, *Macroeconomic Perspectives on the Equity Premium Puzzle*

548 B. PELLOUX, *the Role of Emotions and Social Ties in Public On Good Games: Behavioral and Neuroeconomic Studies*

549 N. YANG, *Markov-Perfect Industry Dynamics: Theory, Computation, and Applications*

550 R.R. VAN VELDHUIZEN, *Essays in Experimental Economics*

551 X. ZHANG, *Modeling Time Variation in Systemic Risk*

552 H.R.A. KOSTER, *The internal structure of cities: the economics of agglomeration, amenities and accessibility*

553 S.P.T. GROOT, *Agglomeration, globalization and regional labor markets: micro evidence for the Netherlands*

554 J.L. MÖHLMANN, *Globalization and Productivity Micro-Evidence on Heterogeneous Firms, Workers and Products*

555 S.M. HOOGENDOORN, *Diversity and Team Performance: A Series of Field Experiments*

556 C.L. BEHRENS, *Product differentiation in aviation passenger markets: The impact of demand heterogeneity on competition*

557 G. SMRKOLJ, *Dynamic Models of Research and Development*

558 S. PEER, *The economics of trip scheduling, travel time variability and traffic information*

559 V. SPINU, *Nonadditive Beliefs: From Measurement to Extensions*

560 S.P. KASTORYANO, *Essays in Applied Dynamic Microeconometrics*

561 M. VAN DUIJN, *Location, choice, cultural heritage and house prices*

562 T. SALIMANS, *Essays in Likelihood-Based Computational Econometrics*

563 P. SUN, *Tail Risk of Equidity Returns*

564 C.G.J. KARSTEN, *The Law and Finance of M&A Contracts*

565 C. OZGEN, *Impacts of Immigration and Cultural Diversity on Innovation and Economic Growth*

566 R.S. SCHOLTE, *The interplay between early-life conditions, major events and health later in life*

567 B.N. KRAMER, *Why don't they take a card? Essays on the demand for micro health insurance*

568 M. KILIÇ, *Fundamental Insights in Power Futures Prices*

569 A.G.B. DE VRIES, *Venture Capital: Relations with the Economy and Intellectual Property*

570 E.M.F. VAN DEN BROEK, *Keeping up Appearances*

571 K.T. MOORE, *A Tale of Risk: Essays on Financial Extremes*

572 F.T. ZOUTMAN, *A Symphony of Redistributive Instruments*

573 M.J. GERRITSE, *Policy Competition and the Spatial Economy*

574 A. OPSCHOOR, *Understanding Financial Market Volatility*

575 R.R. VAN LOON, *Tourism and the Economic Valuation of Cultural Heritage*

576 I.L. LYUBIMOV, *Essays on Political Economy and Economic Development*

577 A.A.F. GERRITSEN, *Essays in Optimal Government Policy*

578 M.L. SCHOLTUS, *The Impact of High-Frequency Trading on Financial Markets*

579 E. RAVIV, *Forecasting Financial and Macroeconomic Variables: Shrinkage, Dimension reduction, and Aggregation*

580 J. TICHEM, *Altruism, Conformism, and Incentives in the Workplace*

581 E.S. HENDRIKS, *Essays in Law and Economics*

582 X. SHEN, *Essays on Empirical Asset Pricing*

583 L.T. GATAREK, *Econometric Contributions to Financial Trading, Hedging and Risk Measurement*

584 X. LI, *Temporary Price Deviation, Limited Attention and Information Acquisition in the Stock Market*

585 Y. DAI, *Efficiency in Corporate Takeovers*

586 S.L. VAN DER STER, *Approximate feasibility in real-time scheduling: Speeding up in order to meet deadlines*