

Throughput Maximization of Real-Time Scheduling with Batching

Amotz Bar-Noy*

Sudipto Guha†

Yoav Katz‡

Joseph (Seffi) Naor‡

Baruch Schieber§

Hadas Shachnai¶

Abstract

We consider the following scheduling with batching problem that has many applications, e.g., in multimedia-on-demand and manufacturing of integrated circuits. The input to the problem consists of n jobs and k parallel machines. Each job is associated with a set of time intervals in which it can be scheduled (given either explicitly or non-explicitly), a weight, and a family. Each family is associated with a processing time. Jobs that belong to the same family can be batched and executed together on the same machine. The processing time of each batch is the processing time of the family of jobs it contains. The goal is to find a non-preemptive schedule with batching that maximizes the weight of the scheduled jobs. We give constant factor (4 or $4 + \varepsilon$) approximation algorithms for two variants of the problem, depending on the precise representation of the input. When the batch size is unbounded and each job is associated with a time window in which it can be processed, these approximation ratios reduce to 2 and $2 + \varepsilon$, respectively. We also show exact algorithms for several special cases.

1 Introduction

Usually, in scheduling problems, exclusiveness is one of the basic constraints; that is, two jobs cannot be scheduled on the same machine at the same time. In this paper, we explore models that need not obey this constraint and even benefit from batching several jobs of the same type together. Such models have

many applications as detailed below. One example is scheduling clients in a multimedia-on-demand (MOD) system. Each client requests a specific video program at several possible times. When several clients are willing to view the same program at the same time, their requests can be batched and satisfied by a single transmission.

The above scenario can be formulated as follows. The input to the problem consists of n jobs (clients) and k parallel machines (channels). For each job (client) we are given a *weight* (revenue) and a *processing time*. Also, for each job, we either have a release time and a due date that define a window of time in which it can be processed, or an (explicit) set of possible time intervals in which the job can be processed, or a combination of both. The jobs are partitioned into *families* (all the clients requesting the same program), and all jobs belonging to a particular family have the same processing time. Jobs that belong to the same family can be *batched* and executed together, in the same time that it takes to execute a single job from that family. The number of jobs (from the same family) that can be batched together can be either *bounded* or *unbounded*. The goal is to find a feasible non-preemptive schedule with batching that maximizes the weight (revenue) of the scheduled jobs. Such scheduling problems are frequently referred to as *real-time* scheduling problems with *batching of incompatible families* (*f-batch* for short), and the objective of maximizing the value of completed jobs is frequently referred to as *throughput*. In the standard $\alpha|\beta|\gamma$ notation for scheduling problems, the problem we consider here is either $P|f\text{-batch}, b, r_j|\sum w_j(1-U_j)$, or $P|f\text{-batch}, r_j|\sum w_j(1-U_j)$, depending on whether the batch size is bounded by a parameter b or unbounded. Both versions of our problem are strongly NP-hard. Indeed, if every family consists of a single job, then our problem is equivalent to maximizing throughput in real-time scheduling when no batching is allowed (see e.g., [BB⁺00]).

We note that when all jobs have the same length and the same release times, with bounded batch size, our problem reduces to a special case of the *class-constrained multiple knapsack* [ST-01] (see below); when

*AT&T Shannon Lab. 180 Park Ave., P.O. Box 971, Florham Park, NJ 07932. E-mail: amotz@research.att.com. This work was done while the author was a member of the Electrical Engineering Department, Tel Aviv University, Tel Aviv 69978, Israel.

†Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104. E-mail: sudipto@cis.upenn.edu. This work was done while the author was at AT&T Research, Florham Park, NJ 07932.

‡Computer Science Department, Technion, Haifa 32000, Israel. E-mail: {katzy@tx, naor@cs, hadas@cs}.technion.ac.il.

§IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598. E-mail: sbar@watson.ibm.com.

¶Currently on leave at Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974.

the batch size is unbounded we get an instance of *maximum weighted matching* in bipartite graphs (which is polynomially solvable).

1.1 Applications

Our problem has numerous applications in activity selection and in resource sharing among jobs with conflicting requirements. We describe applications related to MOD services and production planning.

In a MOD system (see, e.g., [DSS-96] and the introduction in [AGH-95]) clients send requests to view video programs to a centralized video server. The system has a fixed number of channels, through which the programs are transmitted to the clients; using a multicast facility, the server can batch several clients (who wish to view the same program) to use the same communication channel. The server needs to decide which of the requests will be serviced, and in which order, such that revenue is maximized. (We assume that clients must have some degree of “patience”, in order to allow requests to be batched together.) Note that in this application, if the network has unlimited multicast capabilities, the batch size will be unbounded. However, with Internet multicasting, it is reasonable to assume bounded multicasting to assure reliability.

The production of very large-scale integrated circuits (VLSI) involves complex processes. Since orders for the products come with strict delivery times, it is important to optimize the execution of these processes. One of the processes in the wafer fabrication stage is diffusion. This process is long and is often the bottleneck in wafer fabrication. The diffusion is done in a reactor that has the capacity to process several jobs simultaneously. However, due to differences in the chemical nature of the products, jobs of different families cannot be batched together in the diffusion reactor. Suppose that we are given a set of products from different families, each with an arrival time, a due date, and a revenue. Our goal is to schedule the reactor to maximize the total revenue of jobs that meet their deadlines. Note that in this application the batch size is bounded by the size of the diffusion reactor.

Another VLSI related application is scheduling the thermal treatment for Multi-Layer Ceramic (MLC) packaging. The last stage of MLC manufacturing is a thermal treatment that is done in an oven. Due to the length of this process (around 24 hours) and the limited size of the oven, this process is the manufacturing bottleneck. Here, several jobs can be batched together for the process as long as their thermal treatment is the same, and the goal is to maximize the (weighted) throughput of jobs completed before their deadlines.

1.2 Contribution

In this paper, we give constant factor $(4 \text{ or } 4 + \varepsilon)^*$ approximation algorithms for two variants of the problem, depending on the input instance (discrete vs. continuous time). When the batch size is unbounded and each job is associated with a time window in which it can be processed, the approximation factor reduces to $2 + \varepsilon$. To the best of our knowledge, this paper gives for the first time approximation algorithms with guaranteed performance bounds for these problems. Our approximation algorithms are based on a nontrivial application of the *local ratio* technique and are not hard to implement. Our technique can be extended to a more general problem (that generalizes also the parallel batch problem described below), where jobs in the same family have different processing times and the processing time of a batch is determined by the longest job in the batch. It can also be generalized to the case where the possible time intervals of the same job have different lengths and weights.

We also give exact polynomial time algorithms for several special cases of the problem. In particular, we show that real time scheduling with batching is polynomially solvable in the following classes of instances: (i) the jobs offer small *slacks* (the slack of a job is the maximal possible delay from the time it is released until it is scheduled); (ii) the number of families is fixed, and (iii) the number of release times and due dates, as well as the batch size are bounded by a constant, and all jobs have the same length and weight. Note that if we slightly modify the third class of instances to allow jobs from different families to have different lengths, the problem becomes NP-complete, even when the release dates and due dates can take only two values each [GJ-79].

1.3 Related Work

Maximizing the throughput in real-time scheduling without batching was studied extensively in [S-99, BB⁺00, BD-00, BG⁺01, COR-01]. All of these papers focused on the case where jobs specify more than one time interval in which they can be performed (in either a discrete or a continuous fashion). This model captures many applications, e.g., scheduling a space mission, bandwidth allocation, and communication in a linear network. The approximation factor obtained is 2 (and $2 + \varepsilon$ for the continuous case). Very recently, the approximation factor was improved in [COR-01] to $(e/(e-1) + \varepsilon)$ for the unweighted version of the problem, where ε is some constant.

*We define the approximation ratio as the ratio of the optimum solution to the solution given by the algorithm. By this definition the approximation factor of any algorithm is always at least 1.

In the *parallel batch processing* (*p-batch*) model, each machine can batch several jobs to run in parallel. In this model, all the jobs are assumed to belong to the same family and thus any group of jobs can be batched together. However, jobs may have different lengths. A batch is completed when the longest job in the batch is completed. Brucker et al. [BG⁺00] showed that in this model the problem $1|p\text{-batch}|\sum w_j(1-U_j)$ is strongly NP-hard. Baptiste [B-00] showed that when all jobs have the same length the bounded batch case is solvable in $O(n^8)$ steps, even when the jobs are released at different times. Our exact algorithm for a fixed number of families has improved complexity when applied to the bounded *p-batch* problem.

Batching with incompatible job families was studied previously in the Operations Research literature, however the objective functions were different than ours. Specifically, the measures considered in these works were the weighted sum of completion times or the makespan (see e.g. [U-95, DN-99, AW-01]). The paper [MU-98] presented exact and heuristic algorithms for the problem of family batching with the objective of minimizing total tardiness. Devpura et al. [DF⁺99] examined the case where the objective is to minimize total weighted tardiness. This problem is NP-hard even in the case where there are no release dates; their paper presents various heuristic approaches for tackling this problem.

Uzsoy [U-95] studied the feasibility version of our problem, in which we need to determine whether all jobs can be scheduled and meet their due dates. The paper shows that with no release times the feasibility problem is solvable in $O(n \log n)$ steps, using a variant of the *earliest due date* (EDD) algorithm.

Finally, as mentioned above, when all families have the same execution times and no release dates, our problem can be reduced to the class-constrained multiple knapsack. Indeed, we can associate each time slot with a knapsack of capacity b , in which we can pack unit sized item (jobs) from a single class (family). An item can be packed when the corresponding job is available. A greedy algorithm proposed in [CK-00] can be adapted to yield a 2-approximation ratio for this problem.

1.4 Organization of the Paper

In Section 2 we introduce the local ratio technique and some notation. In section 3, we describe the general local ratio algorithm, and in Section 4, we consider the unbounded case. In Section 5, we give polynomial time exact algorithms for several special cases. Due to space constraints, we omit the description of our results for the more generalized case in which the possible time intervals of the same job have different length and

weight.

2 Preliminaries

2.1 The Local Ratio Technique

Our algorithms are based on the local ratio technique, developed by Bar-Yehuda and Even [BE-85], and later extended by Bafna, Berman and Fujito [BBF-95]. We describe below the maximization variant of the local ratio technique required for our algorithm (see [BB⁺00]).

Let $\mathbf{w} \in \mathbb{R}^n$ be a weight vector, and let \mathcal{F} be a set of feasibility constraints on vectors $\mathbf{x} \in \mathbb{R}^n$. A vector $\mathbf{x} \in \mathbb{R}^n$ is a *feasible solution* to a given problem $(\mathcal{F}, \mathbf{w})$ if it satisfies all of the constraints in \mathcal{F} . The *value* of a feasible solution \mathbf{x} is the inner product $\mathbf{w} \cdot \mathbf{x}$. A feasible solution is *optimal* for a maximization problem if its value is larger than or equal to the value of all feasible solutions. A feasible solution \mathbf{x} is a ρ -*approximate* solution, or simply a ρ -*approximation*, if $\mathbf{w} \cdot \mathbf{x} \geq \frac{1}{\rho} \cdot \mathbf{w} \cdot \mathbf{x}^*$, where \mathbf{x}^* is an optimal solution. An algorithm is said to have a *performance guarantee* of ρ , if it always computes ρ -approximate solutions.

THEOREM 2.1. (Local Ratio) *Let \mathcal{F} be a set of constraints and let \mathbf{w} , \mathbf{w}_1 , and \mathbf{w}_2 be weight vectors such that $\mathbf{w} = \mathbf{w}_1 + \mathbf{w}_2$. Then, if \mathbf{x} is a ρ -approximate solution with respect to $(\mathcal{F}, \mathbf{w}_1)$ and with respect to $(\mathcal{F}, \mathbf{w}_2)$, then \mathbf{x} is a ρ -approximate solution with respect to $(\mathcal{F}, \mathbf{w})$.*

Proof. Let \mathbf{x}^* , \mathbf{x}_1^* , \mathbf{x}_2^* be optimal solutions for $(\mathcal{F}, \mathbf{w})$, $(\mathcal{F}, \mathbf{w}_1)$, and $(\mathcal{F}, \mathbf{w}_2)$ respectively. Then $\mathbf{w} \cdot \mathbf{x} = \mathbf{w}_1 \cdot \mathbf{x} + \mathbf{w}_2 \cdot \mathbf{x} \geq \frac{1}{\rho} \cdot \mathbf{w}_1 \cdot \mathbf{x}_1^* + \frac{1}{\rho} \cdot \mathbf{w}_2 \cdot \mathbf{x}_2^* \geq \frac{1}{\rho} \cdot (\mathbf{w}_1 \cdot \mathbf{x}^* + \mathbf{w}_2 \cdot \mathbf{x}^*) = \frac{1}{\rho} \cdot \mathbf{w} \cdot \mathbf{x}^*$ \square

The Local Ratio Theorem is usually applied in the following way. Given a problem defined in the above formulation, we find a decomposition of \mathbf{w} into $\mathbf{w}_1 + \mathbf{w}_2$ with the property that every *maximal* solution is a ρ -approximate solution with respect to $(\mathcal{F}, \mathbf{w}_1)$. We solve the problem recursively with respect to $(\mathcal{F}, \mathbf{w}_2)$. Then, we extend the ρ -approximate solution with respect to $(\mathcal{F}, \mathbf{w}_2)$ found in the recursion to a maximal solution. The resulting solution is a ρ -approximate solution with respect to $(\mathcal{F}, \mathbf{w}_1)$ and with respect to $(\mathcal{F}, \mathbf{w}_2)$, thus it is a ρ -approximate solution with respect to $(\mathcal{F}, \mathbf{w})$. In most applications of the local ratio technique, the most involved part is finding the decomposition of the weight function.

The Local Ratio Theorem applies to all problems in the above formulation. Note that \mathcal{F} can include arbitrary feasibility constraints and not just linear, or linear integer, constraints. Nevertheless, all successful applications of the local ratio technique to date involve

problems in which the constraints are either linear or linear integer, and this is also the case for the problems treated herein.

2.2 Definitions and Notation

Suppose that n jobs $\{J_1, \dots, J_n\}$ need to be scheduled on a set of k machines. There are F different job families; all the jobs in a family $f \in [1..F]$ have the same processing time, p_f . Each job J_j belongs to a family f_j has weight w_j , and processing time p_j . A problem instance may be either *discrete* or *continuous*. In a discrete instance, for each job we have an explicit list of the time intervals in which it can be scheduled. In a continuous instance, each job comes with a release date, r_j , and a due date, d_j , defining a time window in which the job can be processed (and which is typically larger than the processing time). We note that our algorithm applies also to the more general case where each job is associated with a number of possible time windows. However, to keep the presentation simple we consider the case of a single time window per job.

We distinguish between *bounded* and *unbounded* batching. In the case of unbounded batching, any number of jobs can be batched together on each of the machines, as long as they belong to the same family. In the case of bounded batching, each of the machines can process the jobs in batches of at most b jobs, with the restriction that all jobs in a batch belong to the same family. Our results apply also for the case in which each family f has a different bound b_f . To simplify the presentation we assume a single bound b .

A *job instance* is a job and a feasible time interval in which it can be executed. A *batch instance* is a set of at most b job instances; all belong to the same family and have the same execution time interval.

For any given time t , and for each family f , let $\mathcal{J}_{f,t}$ be the set of jobs from family f that can start at time t . Note that the possible batch instances of family f that can start at t are all subsets of $\mathcal{J}_{f,t}$ of size at most b . For a batch instance B , let $f(B)$ be the family of the jobs in B , $t(B)$ be the starting time of the batch B and $p(B)$ be the processing time of the jobs in batch B ($p(B) = p_{f(B)}$). Denote by $I(B)$ the time interval of this batch instance, i.e., $I(B) = [t(B), t(B) + p(B)]$; $J(B)$ is the set of jobs in B , and $w(B)$ is the sum of the weights of the jobs in B , i.e., $w(B) = \sum_{j \in J(B)} w_j$.

We say that two batch instances B and B' are *simultaneous* if $I(B) = I(B')$. Two batch instances B and B' *conflict in time* if $I(B)$ and $I(B')$ intersect. Two batch instances B and B' *conflict in jobs* if $J(B)$ and $J(B')$ intersect. Two batch instances *conflict* if they conflict in either time or jobs. A batch instance B' is *contained* in batch instance B if $J(B') \subseteq J(B)$. A

batch instance B is an *extension* of a batch instance B' , denoted $B' \preceq B$, if B and B' are simultaneous and B contains B' ; B is a *proper extension* of B' , $B' \prec B$, if $B' \preceq B$ and $J(B') \subset J(B)$. Conversely, a batch instance B is a (proper) *reduction* of a batch instance B' , if B' is a (proper) extension of B . Note that a batch instance B is both an extension and a reduction of itself.

A *feasible* schedule consists of a set of batch instances \mathcal{B} such that (i) all the batch instances in \mathcal{B} do not conflict in jobs, and (ii) the batch instances in \mathcal{B} can be partitioned into k subsets of non-conflicting batch instances, where k is the number of available machines. The objective is to find a feasible schedule that maximizes the overall weight of scheduled batches. We call this problem *real-time scheduling with batching*.

3 The General Algorithm

We present the approximation algorithm for a discrete instance and a single machine. Later, we show how to extend it to other cases.

We start with a generic scheme based on the local ratio technique. This scheme is recursive following the generic description given in the previous section. In the scheme we consider a slightly more general problem, where instead of having a weight per job, there is a weight per batch instance. The goal is to schedule a set of non conflicting batch instances with maximum weight. Clearly, this problem is a generalization of the original problem, in which the weight of a batch instance B is the sum of the weights of the jobs in $J(B)$. Note that the number of batch instances may be super polynomial. To keep the size of the input polynomial, we assume that the weights of the batch instances are given implicitly, as described later in the polynomial time implementation.

In the scheme we consider batch instances with negative weights. Although the initial weights can be assumed to be positive, weights may become negative during the recursive calls. Also, we note that during the recursive calls, some batch instances are deleted. We refer to batch instances that were not deleted as *available*.

The general Scheme:

1. Delete all batch instances with non-positive weight.
2. If no batch instances remain, return the empty schedule. Otherwise, proceed to the next step.
3. Select a batch instance \tilde{B} and decompose w by $w = w_1 + w_2$. The exact choice of \tilde{B} and the decomposition of w is described below.
4. Solve the problem recursively using w_2 as the weight function. Let \mathcal{S}' be the schedule returned.

5. Turn \mathcal{S}' into a \tilde{B} -maximal schedule \mathcal{S} . The definition of a \tilde{B} -maximal schedule and how to turn \mathcal{S}' into a \tilde{B} -maximal schedule is described below.

We now elaborate on the steps of the algorithm, and analyze the quality of the solution it outputs. We start with the definition of a B -maximal solution.

DEFINITION 3.1. *A schedule \mathcal{S} is B -maximal if it contains a (possibly empty) batch $B' \preceq B$ such that B' cannot be replaced in \mathcal{S} by any other batch B'' , where $B' \prec B'' \preceq B$, without violating feasibility.*

Observe that if a B -maximal schedule \mathcal{S} does not contain all the jobs in $J(B)$, then it must contain a batch instance that conflicts with B in time and yet it is not a reduction of B .

In the last step of the algorithm, if \mathcal{S}' is not a \tilde{B} -maximal schedule we make it \tilde{B} -maximal by adding the batch instance that is a reduction of \tilde{B} and which consists of all the jobs in $J(\tilde{B})$ that were not scheduled in \mathcal{S}' . Later, we observe that this batch instance is guaranteed to be available.

We are going to choose \tilde{B} and decompose w such that the following two conditions hold.

The w_1 condition: Every \tilde{B} -maximal schedule is a 4-approximation with respect to w_1 .

The w_2 condition: For every available batch instance B that is a reduction of \tilde{B} , $w_2(B) = 0$.

PROPOSITION 3.1. *Suppose that the method for choosing \tilde{B} and decomposing the weight function satisfies both the w_1 and the w_2 conditions. Then, the schedule \mathcal{S} returned by the algorithm is a 4-approximation.*

Proof. Clearly, the first step in which instances of non-positive weight are deleted does not change the optimal value. Thus, it is sufficient to show that \mathcal{S} is a 4-approximation with respect to the remaining instances. The proof is by induction on the number of recursive calls. At the basis of the recursion, the schedule returned is optimal (and hence a 4-approximation), since no instances remain. For the induction step, assume that \mathcal{S}' is a 4-approximation solution with respect to w_2 . Note that \mathcal{S} is either the same as \mathcal{S}' or is given by adding an available batch instance that is a reduction of \tilde{B} to \mathcal{S}' . It follows from the w_2 condition that \mathcal{S} is a 4-approximation with respect to w_2 . Since \mathcal{S} is \tilde{B} -maximal, it follows from the w_1 condition that it is also a 4-approximation with respect to w_1 . Thus, by the Local Ratio Theorem, it is a 4-approximation with respect to w . \square

It remains to specify how to determine \tilde{B} and the decomposition of the weight function. The choice of \tilde{B} is done by selecting a batch instance with minimum end-time among all batch instances. Among all batch instances with the minimum end-time we select a batch instance \tilde{B} such that \tilde{B} is maximal with respect to extension; i.e., there are no other simultaneous batch instances B' such that $J(\tilde{B}) \subset J(B')$. Among all these batch instances we select a batch instance with the maximum weight, breaking ties arbitrarily.

We now define the decomposition $w = w_1 + w_2$. Recall that, initially, for each batch instance B , $w(B) = \sum_{j \in J(B)} w_j$. During the recursive calls, the weight of each batch instance is changed and will be given by

$$w(B) = \sum_{j \in J(B)} u_{j,B} - \Delta_B.$$

Initially, $u_{j,B} = w_j$ and $\Delta_B = 0$, for all batch instances B . Thus, the initial weight of a batch instance B is the sum of the original weights of the jobs in $J(B)$.

Consider now a recursive call defined by Δ_B and $u_{j,B}$, for all batch instances B and jobs $j \in J(B)$. These quantities determine the weight $w(B)$ of each batch instance B . We show the decomposition of the weight $w(B)$ into $w_1(B) + w_2(B)$. It suffices to define $w_2(B)$ which is determined by the updated values of $u_{j,B}$ and Δ_B . The value of $w_1(B)$ is therefore the amount of weight added or subtracted from $w(B)$ to define $w_2(B)$.

The following quantities are defined based on the values of $u_{j,B}$ before the update. For a batch instance B ,

- let $u(B)$ be $\sum_{j \in J(B)} u_{j,B}$, and
- let $m(B)$ be $\max\{u(B')\}$ over all the batch instances B' that are extensions of B .

Later, we show how to maintain these quantities implicitly in polynomial time.

The quantities Δ_B and $u_{j,B}$ are updated only for batch instances that conflict with \tilde{B} . We distinguish between three types of such conflicting batch instances:

1. For each batch instance B that conflicts with \tilde{B} only in jobs (i.e., a batch instance B for which $I(\tilde{B})$ and $I(B)$ do not intersect, but $J(\tilde{B})$ and $J(B)$ intersect), Δ_B remains unchanged. For each $j \in J(B) \cap J(\tilde{B})$, $u_{j,B}$ is decremented by its “relative share” of $w(\tilde{B})$; that is,

$$u_{j,B} = u_{j,B} - \frac{u_{j,\tilde{B}}}{u(\tilde{B})} \cdot w(\tilde{B}).$$

As a result, in this case,

$$w_1(B) = \frac{\sum_{j \in J(B) \cap J(\tilde{B})} u_{j,\tilde{B}}}{u(\tilde{B})} \cdot w(\tilde{B}).$$

2. For each batch instance B that is a reduction of \tilde{B} , set

$$\Delta_B = \sum_{j \in B} u_{j,B}$$

and keep $u_{j,B}$ unchanged. Note that this implies that $w_2(B) = 0$ and $w_1(B) = w(B)$.

3. For each batch instance B that is not a reduction of \tilde{B} and which conflicts with \tilde{B} in time, increment Δ_B by

$$\max \left\{ \frac{1}{2}, \frac{u(B)}{m(B)} \right\} \cdot w(\tilde{B}) .$$

In words, consider the ratio of $u(B)$ to $m(B)$: if it is at most half, increment Δ_B by $\frac{1}{2}w(\tilde{B})$, otherwise increment it by $w(\tilde{B})$ multiplied by this ratio. The quantities $u_{j,B}$ are unchanged. Note that the value of $w_1(B)$ is exactly this increment in Δ_B .

To prove the correctness of our algorithm, we need to show that every \tilde{B} -maximal schedule is a 4-approximation with respect to w_1 . For this we show that the optimal weight is at most $2w_1(\tilde{B})$, and that the weight of every \tilde{B} -maximal solution is at least $\frac{1}{2}w_1(\tilde{B})$.

LEMMA 3.1. *The optimal solution with respect to w_1 is at most $2w_1(\tilde{B})$.*

Proof. Consider an optimal solution that consists of a set of batch instances \mathcal{B} . By the definition of w_1 , the weight of all the batch instances in \mathcal{B} that do not conflict with \tilde{B} is zero.

If the optimal solution contains \tilde{B} , then it cannot contain any other batch instance that conflicts with \tilde{B} . It follows that $\sum_{B \in \mathcal{B} \setminus \{\tilde{B}\}} w_1(B) = 0$, and the weight of the optimal solution is $w_1(\tilde{B})$.

Suppose that the optimal solution does not contain \tilde{B} . In this case it may contain batch instances that conflict with \tilde{B} only in jobs, and at most one batch instance \hat{B} that conflicts with \tilde{B} in time. Note that for all batch instances $B \in \mathcal{B}$ that conflict with \tilde{B} only in jobs $\Delta_B = 0$, and thus $w_1(B) = \frac{\sum_{j \in J(B) \cap J(\tilde{B})} u_{j,\hat{B}}}{u(\tilde{B})}$.

Since the sets $J(B)$ for all batch instances $B \in \mathcal{B}$ that conflict with \tilde{B} only in jobs are mutually disjoint, the union of these sets may include at most one copy of each job $j \in \tilde{B}$. Let $H \subseteq J(\tilde{B})$ be the subset of the jobs in $J(\tilde{B})$ included in the union of these sets. The sum of the weights of all these batch instances is bounded by

$$\frac{\sum_{j \in H} u_{j,\hat{B}}}{u(\tilde{B})} \cdot w(\tilde{B}) \leq w(\tilde{B}) .$$

Now, consider the batch instance \hat{B} that conflicts with \tilde{B} in time. If \hat{B} is not a reduction of \tilde{B} then $w_1(\hat{B})$

equals to the increment in $\Delta_{\hat{B}}$ which is bounded by $w_1(\tilde{B})$.

Suppose that \hat{B} is a reduction of \tilde{B} . Since $I(\hat{B}) = I(\tilde{B})$, both \hat{B} and \tilde{B} start to conflict in time with the batch instances chosen in the recursive calls at the same time. Also, following the first such call, both conflict in time with all the batch instances chosen in subsequent recursive calls, until the recursive call in which \tilde{B} is chosen. Note that this implies that $u_{j,\hat{B}} = u_{j,\tilde{B}}$, for all $j \in J(\hat{B}) \subset J(\tilde{B})$. This is because $u_{j,\hat{B}}$ and $u_{j,\tilde{B}}$ are updated in the same way in all recursive calls before \hat{B} and \tilde{B} start to conflict in time with the batch instances chosen in the recursive calls, and are kept unchanged from that point on. It follows that $u(\hat{B}) \leq u(\tilde{B})$. By our choice of \hat{B} , $m(\hat{B}) = u(\hat{B})$ and $m(\tilde{B}) = u(\tilde{B})$. Hence, $\Delta_{\hat{B}} \geq \frac{u(\hat{B})}{u(\tilde{B})} \Delta_{\tilde{B}}$. It follows that $w_1(\hat{B}) = u(\hat{B}) - \Delta_{\hat{B}} \leq u(\tilde{B}) - \Delta_{\tilde{B}} = w_1(\tilde{B})$. \square

LEMMA 3.2. *The weight of every \tilde{B} -maximal solution is at least $\frac{1}{2}w_1(\tilde{B})$.*

Proof. To prove this claim we distinguish between three cases.

CASE 1: If \tilde{B} belongs to the solution, the claim is clearly true.

CASE 2: The solution contains a batch instance B that is not a reduction of \tilde{B} and conflicts with \tilde{B} in time. By our construction, $w_1(B)$ equals to the reduction in Δ_B , which is at least $\frac{1}{2}w_1(\tilde{B})$.

CASE 3: Suppose that the solution does not contain any batch instance that is not a reduction of \tilde{B} and conflicts with \tilde{B} in time. The solution may schedule some of the jobs in $J(\tilde{B})$ in batch instances that do not conflict with \tilde{B} in time. Let $H \subseteq J(\tilde{B})$ be the set of these jobs.

CASE 3.1: $\sum_{j \in H} u_{j,\hat{B}} \geq \frac{1}{2}u(\tilde{B})$. Note that the sum of the w_1 -weights taken over all batch instances that contain a job from H is

$$\frac{\sum_{j \in H} u_{j,\hat{B}}}{u(\tilde{B})} w_1(\tilde{B}) \geq \frac{1}{2} w_1(\tilde{B}) .$$

CASE 3.2: $\sum_{j \in H} u_{j,\hat{B}} < \frac{1}{2}u(\tilde{B})$. Consider the batch instance B which is a reduction of \tilde{B} and consists of the jobs in $J(\tilde{B}) \setminus H$. To make the solution \tilde{B} -maximal, the batch instance B has to be added to the solution. As in the proof of Lemma 3.1 $u_{j,B} = u_{j,\hat{B}}$, for all $j \in J(B) \subset J(\tilde{B})$. It follows that $u(B) > \frac{1}{2}u(\tilde{B}) = \frac{1}{2}m(B)$, and thus $\Delta_B = \frac{u(B)}{m(B)} \Delta_{\tilde{B}}$. We get that

$$\begin{aligned} w_1(B) &= u(B) - \Delta_B \\ &= \frac{u(B)}{u(\tilde{B})} \left(u(\tilde{B}) - \Delta_{\tilde{B}} \right) \end{aligned}$$

$$\begin{aligned}
&= \frac{u(B)}{u(\tilde{B})} w_1(\tilde{B}) \\
&> \frac{1}{2} w_1(\tilde{B}) .
\end{aligned}$$

Finally, the batch B is guaranteed to be available since $\frac{1}{2}w_1(\tilde{B}) > 0$. \square

3.1 Polynomial-time implementation

We now show that the algorithm can be implemented in polynomial time. To this end, we need to show that the number of recursive calls is polynomial and that the bookkeeping can be done in polynomial time. During the run of the algorithm we say that the algorithm *processed* time t , if the minimum end-time of the batch instances currently considered by the algorithm is later than t . Each recursive call of the algorithm is associated with a *call time* which is the end-time of the batch instance \tilde{B} in that call.

We first show how to maintain $u_{j,B}$ and Δ_B . Consider a batch instance B , the following observations hold by definition.

1. $\Delta_B = 0$ as long as the algorithm has not processed time $t(B)$ (the start point of B).
2. Δ_B is the same for all simultaneous batch instances B from the same family for which $u(B) = m(B)$.
3. $u_{j,B}$ is identical for all batch instances B such that $t(B)$ is later than the current call time \tilde{t} .
4. $u_{j,B}$ remains unchanged after the recursive call with the latest call time that ends before $t(B)$.

For a job $j \in J$, to maintain $u_{j,B}$ we maintain one single set of values for all batch instances B such that $t(B)$ is later than the current call time \tilde{t} . We maintain $u_{j,B}$ for all batch instances B such that $t(B) \leq \tilde{t} \leq t(B) + p(B)$ implicitly. This can be done since $u_{j,B}$ can be computed using $u_{j,B'}$ where $t(B') > \tilde{t}$, which is maintained. $u_{j,B}$ is $u_{j,B'}$ plus the total reductions of $u_{j,B'}$ in all the recursive calls with call times between $t(B)$ and \tilde{t} . Since the number of these calls is polynomial, this can be computed in polynomial time.

To maintain (implicitly) Δ_B for all batch instances B such that $t(B) \leq \tilde{t} \leq t(B) + p(B)$, we note that Δ_B is $u(B)/m(B)$ time the sum of $w(\tilde{B})$ in all recursive calls with call times in the interval $[t(B), \tilde{t}]$. Again, since the number of these calls is polynomial, this can be computed in polynomial time. We remark that $m(B)$ can be computed by checking if $|J(B)| < b$ and in case this is true $m(B)$ is given by adding to $u(B)$ the weight

of (at most) $b - |J(B)|$ jobs not in $J(B)$ from the family $f(B)$ with the maximum value of $u_{j,B}$.

Consider one of the recursive calls. Recall that \tilde{B} is the batch instance having the maximum weight among all batch instances having minimum end-point (denoted by \tilde{t}). Time \tilde{t} is easy to compute since the input is discrete. To find \tilde{B} we consider all the jobs that can be finished at time \tilde{t} . We have one candidate for \tilde{B} from each family. Consider a job family f . We first find the maximum $m(B)$ for all batch instances from family f . This is the sum of (up to) b jobs from family f that can end at \tilde{t} with the maximum value of $u_{j,B}$ (which is the same for all batches B from f with this end-time). The candidate is a batch instance B with $u(B)$ equals this maximum.

Finally, we remark that since at each recursive call we delete at least one batch instance that is maximal with respect to extension (and its reductions), the number of recursive calls is polynomial.

3.2 Continuous Input

We now explain how to handle continuous input, i.e., the case where batches can start at any point on the time line. Our exposition follows [BB⁺00]. The idea is to operate on whole windows at a time, rather than modify the parameters of individual batch instances. At each iteration we delete all windows whose batch instances have non-positive profit, and find a batch instance \tilde{B} with earliest end-time among the remaining batch instances. The invariant that we maintain is that the parameters of all job instances belonging to the same window are the same. This requires splitting windows; it is easy to see that the points at which the time windows may be split are all of the form: start-time of some window plus a finite sum of lengths of batch instances (not necessarily of the same family). Since any such point can be no greater than the maximum end-time of an instance, there are only finitely many such points. Thus, this implementation always halts in finite, if super-polynomial, time. In order to attain polynomial running time we trade accuracy for speed. For any fixed ε , $0 < \varepsilon < 1$, we modify the algorithm as follows. Whenever \tilde{B} is chosen such that $w(\tilde{B})$ has dropped below $\varepsilon \cdot \sum_{j \in J(\tilde{B})} w_j$ we simply delete the window containing \tilde{B} and do not alter any other parameters. It can be shown that the running time of the algorithm with this change is $O(1/\varepsilon)$ times the running time of the discrete input algorithm. The approximation factor obtained degrades by an additive factor of ε , to $4 + \varepsilon$. The reader is referred to [BB⁺00] for more details.

3.3 Multiple Machines

The way to handle multiple machines also follows [BB⁺00]. The recursive calls are the same as for the single machine case, the only change is in the value of $w_1(B)$ for every batch instance B that is not a reduction of \tilde{B} that conflicts with \tilde{B} in time. The increment of Δ_B for these batch instances is $\frac{1}{k} \max\{\frac{1}{2}, \frac{U(B)}{M(B)}\}w(\tilde{B})$; that is, $\frac{1}{k}$ of the increment in the single machine case, where k is the number of machines. Similar to the technique described in [BB⁺00] and following the proof above, it can be shown that after this modification the optimal weight is at most $2w_1(\tilde{B})$, and that the weight of every \tilde{B} -maximal solution is at least $\frac{1}{2}w_1(\tilde{B})$.

4 Unbounded Batching

We describe here how to obtain improved approximation factors for unbounded batching. For continuous input the improved factor is $2 + \varepsilon$. For discrete input the improved factor is 2, however, we need to make the following assumption. For every job J_j , if $[t_1, t_2]$ and $[t_3, t_4]$, where $t_1 < t_3$, are feasible time intervals for J_j ($t_4 - t_3 = t_2 - t_1 = p_j$), then if $[t', t'']$, where $t_1 < t' < t_3$, is feasible for any job in the family f_j , it is also feasible for J_j . We present the algorithm for the case of a single machine and discrete input. The extensions to continuous input, multiple machines, and all implementation issues are similar to the general algorithm. The algorithm follows the framework of the generic scheme. The only difference is in the way the weight function w_1 is computed and in the transformation of \mathcal{S}' to \mathcal{S} .

We consider the jobs in $J(\tilde{B})$ in descending order of deadline: j_1, j_2, \dots . Let z be the maximum index such that $\sum_{i=1}^z u_{j_i, \tilde{B}} \leq w(\tilde{B})$. Since Δ_B can be greater than zero, it follows that z is not necessarily the size of $J(\tilde{B})$. Let

$$\delta = w(\tilde{B}) - \sum_{i=1}^z u_{j_i, \tilde{B}}.$$

Note that $\delta \neq 0$ only if $\sum_{i=1}^z u_{j_i, \tilde{B}} < w(\tilde{B})$. We change the definition of \tilde{B} -maximal solution to be a solution that schedules j_1, \dots, j_{z+1} . To define the decomposition we define w_2 by showing how $u_{j,B}$ and Δ_B are updated. These quantities will be updated only for batch instances that conflict with \tilde{B} .

1. Suppose that batch instance B conflicts with \tilde{B} only in jobs. For each job $j \in \{j_1, \dots, j_z\} \cap J(B)$, set $u_{j,B}$ to zero. If $j_{z+1} \in J(B)$ then decrement $u_{j_{z+1}, B}$ by δ . The value of Δ_B remains unchanged.
2. For each batch instance B that is not a reduction of \tilde{B} that conflicts with \tilde{B} in time increment Δ_B by $w(\tilde{B})$.

3. For each batch instance B that is a reduction of \tilde{B} , set $w_1(B) = w(B)$.

Note that in the recursive call, the weight of jobs $\{j_1, \dots, j_z\}$ is zero and thus we may assume that they are not present. To turn the schedule \mathcal{S}' into a \tilde{B} -maximal solution we do the following. If \mathcal{S}' does not schedule any jobs in $J(\tilde{B})$, we add \tilde{B} to \mathcal{S}' to form \mathcal{S} . Otherwise, \mathcal{S}' scheduled some jobs in $J(\tilde{B})$. Suppose that \mathcal{S}' contains a batch B such that $J(B) \cap J(\tilde{B}) \neq \emptyset$. Since b is unbounded and $t(B) + p(B)$ is guaranteed to be earlier than the deadlines of $\{j_1, \dots, j_z\}$ we may extend B to include these jobs as well. For the same reason we may extend B to include j_{z+1} as well, in case it was not scheduled already. This may only increase the value of the solution relative to w_2 and make it \tilde{B} -maximal.

It is easy to see that in the problem with respect to w_1 the optimal weight is at most $2w_1(\tilde{B})$. We prove that the weight of every \tilde{B} -maximal solution is at least $w_1(\tilde{B})$. The 2-approximation follows.

LEMMA 4.1. *The weight of every \tilde{B} -maximal solution is at least $w_1(\tilde{B})$.*

Proof. Note that the \tilde{B} -maximal solution would never schedule a batch instance that is a reduction of \tilde{B} . The claim is clearly true in case batch instance \tilde{B} is in the solution. Otherwise, all jobs $\{j_1, \dots, j_{z+1}\}$ are scheduled in batch instances that do not conflict \tilde{B} in time. Let \mathcal{B} be the set of batch instances in which these jobs are scheduled. By our construction $\sum_{B \in \mathcal{B}} w_1(B) = w_1(\tilde{B})$. \square

5 Exact Algorithms

5.1 Small Slacks

Consider the case where for any job J_j , $s_j = d_j - (r_j + p_j) < p_j$. Then our problem can be reduced to *maximum weighted k -colorable subgraph* on interval graphs. For each family f and any time t , we define an interval whose weight is the maximal possible weight of a batch from family f that can be scheduled at t . Since each of the jobs has a small slack, if we schedule at time t a batch that contains the job J_j , then clearly J_j is not contained in the next batch of f (since $d_j < t + 2p_f$). Thus, the problem for k parallel machines is reduced to the maximum weight k -colorable graph problem on the resulting instance of interval graph, whose size is polynomial in the sum of the slack times. We conclude

THEOREM 5.1. *The $p|f - \text{batch}, s_j < p_j| \sum w_j(1 - U_j)$ can be solved in weakly polynomial time.*

5.2 Fixed Number of Families

When the number of families is fixed, our problem can be solved using dynamic programming (DP). For bounded batch size we have the following result.

THEOREM 5.2. *The problem $1|f - \text{batch}, b, r_j, F = \text{const}|\sum w_j(1 - U_j)$ can be solved optimally in $O(n^{F^2+3F+3} \log n)$ steps.*

Our algorithm relies on the following structure of optimal schedules. (We omit the proof.)

PROPOSITION 5.1. *There exists an optimal schedule with the following property. Let B be a scheduled batch instance and suppose that $d(B)$ is the maximum due date among all jobs in $J(B)$, then no job from family $f(B)$ with release date $r < t(B)$ and due date $d < d(B)$ is scheduled after time $t(B)$.*

Let $W(t, \tau_1, \dots, \tau_F, \delta_1, \dots, \delta_F)$ be the maximal weight of a schedule which starts at time t , given that the last batch of jobs from family f , $1 \leq f \leq F$, was scheduled at time $\tau_f < t$ and δ_f is the largest due date of any scheduled job from family f . Denote by $\text{nextarrival}(t) = \min_{r_j > t} r_j$. Then $\alpha_t = W(\text{nextarrival}(t), \tau_1, \dots, \tau_F, \delta_1, \dots, \delta_F)$ is the maximal weight of a schedule that starts at the time of the first arrival of some job after t , with τ_f, δ_f defined as above, $1 \leq f \leq F$.

Given a partial schedule that ends by the time t , let $\delta'_f > \delta_f$ be the due date of some job from family f . Recall that $\mathcal{J}_{f,t}$ is the collection of subsets of jobs from family f that can be scheduled at time t . Then, by Proposition 5.1, we can define

$$\mathcal{J}_{f,t} = \left\{ B \mid |B| \leq b, \forall J_j \in J(B) f_j = f, \right. \\ \left. r_j \leq t, d_j \geq t + p_f, d_j \leq \delta'_f, \right. \\ \left. (r_j > \tau_f \text{ or } \delta_f < d_j) \right\}.$$

The maximum weight of any batch $B \in \mathcal{J}_{f,t}$ is given by $\text{maxweight}(f, t, \tau_f, \delta_f, \delta'_f) = \max_{B \in \mathcal{J}_{f,t}} \{\sum_{J_j \in J(B)} w_j\}$.

Suppose that we schedule a batch from f at time t , then given the values $\tau_1, \dots, \tau_{f-1}, t, \tau_{f+1}, \dots, \tau_F$, and the latest due date of any scheduled job, $\delta_1, \dots, \delta_{f-1}, \delta'_f, \delta_{f+1}, \dots$, we define

$$\beta_{f,\delta'_f} = W(t + p_f, \tau_1, \dots, \tau_{f-1}, t, \tau_{f+1}, \dots, \tau_F, \\ \delta_1, \dots, \delta_{f-1}, \delta'_f, \delta_{f+1}, \dots, \delta_F) \\ + \text{maxweight}(f, t, \tau_f, \delta_f, \delta'_f)$$

to be the sum of the maximal total weight of a schedule that starts at time $t + p_f$, and the maximum weight of a batch from family f scheduled at the time t , in which the maximal due date of any job is δ'_f .

LEMMA 5.1. *$W(t, \tau_1, \dots, \tau_F, \delta_1, \dots, \delta_F)$ can be calculated by the following recursion:*

$$W(t, \tau_1, \dots, \tau_F, \delta_1, \dots, \delta_F) = \max(\alpha_t, \max_{f, \delta'_f} \beta_{f,\delta'_f})$$

Proof. Given that we have scheduled batches up to time t , with the values τ_1, \dots, τ_F , we can proceed in one of the following ways: (i) select a family f , for some $1 \leq f \leq F$, and schedule a batch from f at time t . The batch will contain all the jobs from family f that belong to $\mathcal{J}_{f,t}$. We need to optimize on the weight of the selected batch, taking for each family f all possible values for the maximal due date in the batch, δ'_f ; (ii) move to the next possible batch starting point, which is the time of closest job release time; (iii) if neither of the above is possible, then no more jobs can be scheduled. \square

Proof of THEOREM 5.2: Let DP be a dynamic programming algorithm which calculates W using Lemma 5.1. The weight of an optimal schedule can be found by calculating $W(0, 0, \dots, 0, 0, \dots, 0)$. This is the maximum weight of a schedule starting at time 0, where no batches were scheduled (or alternately, where dummy batches containing jobs with zero weights and zero processing time were scheduled at time 0 for each family). We can obtain the resulting schedule by storing in a table the decision made in each step.

For the complexity of DP, it can be shown that there exists an optimal schedule in which every batch starts at a release date or at the end of another batch. Hence, the possible number of scheduling points is $O(n^{F+1})$, and since the number of due dates is bounded by n , we need to calculate W in at most $O(n^{(F+1)^2+F})$ points.

The optimal batch for a specific scheduling point is found by examining all the possible new maximum due dates: for each we find the set of feasible jobs, sort the jobs by weights, and select up to b jobs with maximal weight from each family. This can be done in $O(n \cdot n \log n)$ steps. Hence the overall running time is $O(n^{(F+1)^2+F} \log n)$. \square

When the batch size is unbounded, we can improve the complexity of DP.

THEOREM 5.3. *The problem $1|f - \text{batch}, r_j, b > n_f, F = \text{const}|\sum w_j(1 - U_j)$ can be solved optimally in $O(n^{(F+1)^2+1})$ steps.*

5.3 Small Number of Release Times and Due Dates

Consider now the case where the number of distinct release dates (R) and due dates (D) is bounded by some constant, the batch size is fixed, and the jobs have the

same (unit) length. We partition the jobs into groups: the jobs in the group (f, k, l) belong to the family f , have the k -th release time and the l -th due date. A naive dynamic programming scheme will run in super-polynomial time, since the possible number of batch instances that can be scheduled at any point of time is large. Therefore we precede our dynamic programming algorithm (DP') by a greedy phase, in which we schedule a maximal number of *full* batches.

The algorithm Reverse-Full-Greedy (RFG) iteratively schedules batches 'backwards': from the maximum due date to time zero. The jobs are scheduled using *time windows*. Let $d_0 = 0, d_1 \leq \dots \leq d_D$ be the set of due dates of the given instance; then, the l -th window is the time interval $[d_{l-1}, d_l)$, $1 \leq l \leq D$. In the l -th window, RFG schedules as many full batches as possible from each group that meet the window's maximal due date, d_l . In doing so, RFG gives higher priority to groups with large release dates. If the l -th window is large enough, then at most $b - 1$ unscheduled jobs will remain from each of the groups; else (i.e., the window becomes full), the remaining jobs from each group are moved to the group from the same family and release date with maximal due date d_{l-1} . By the end of the greedy phase, some of the time windows are "full". We may ignore these windows in the second phase. For each window that is not full we are guaranteed that at most $(b - 1)$ jobs from each group can be scheduled in it.

In the second phase we exhaustively search for the optimal schedule of the remaining jobs in the "holes" that remained in the schedule. Algorithm DP' uses the fact that all families with the same number of remaining jobs in all groups, (f, k, l) for all values of l, k , are equivalent with respect to scheduling possibilities. The detailed algorithm will be given in the full version of the paper. We conclude with the next result.

THEOREM 5.4. *Let $b, D, R \geq 1$ be some constants. Then the problem $1|f - \text{batch}, p_f = 1, b = \text{const}, D = \text{const}, R = \text{const} \left| \sum (1 - U_j) \right.$ can be solved in $O(n^{(b^{D^R} + 1)})$ steps.*

References

- [AGH-95] A. Aggarwal, J. Garay, and A. Herzberg. "Adaptive Video on Demand." In *ESA '95*, 538–553.
- [AW-01] M. Azizoglu and S. Webster. "Scheduling a Batch Processing Machine with Incompatible Job Families." *Computer and Industrial Engineering*, **39**:325–335, 2001.
- [BBF-95] V. Bafna, P. Berman, and T. Fujito. "A 2-approximation Algorithm for the Undirected Feedback Vertex Set Problem." *SIAM J. on Disc. Mathematics*, **12**:289–297, 1999.
- [B-00] P. Baptiste. "Batching Identical Jobs." In *Mathematical Methods of Operations Res.*, **53**: 355–367, 2000.
- [BB⁺00] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber. "A Unified Approach to Approximating Resource Allocation and Scheduling." In *STOC '00*, 735–744.
- [BG⁺01] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. "Approximating the Throughput of Multiple Machines in Real-Time Scheduling." *SIAM Journal on Computing*, **31**:331–352, 2001.
- [BE-85] R. Bar-Yehuda and S. Even. "A Local Ratio Theorem for Approximating the Weighted Vertex Cover Problem." *Annals of Discrete Math.*, **25**:27–46, 1985.
- [BD-00] P. Berman and B. DasGupta. "Multi-Phase Algorithms for Throughput Maximization for Real-Time Scheduling." *Journal of Combinatorial Optimization*, **4**:307–323, 2000.
- [BG⁺00] P. Brucker, A. Gladky, H. Hoogeveen, M. Y. Kovalyov, C. Potts, T. Tautenhahn, and S. L. Van de Velde. "Scheduling a Batching Machine." *Journal of Scheduling*, **1**:31–54, 1998.
- [COR-01] J. Chuzhoy, R. Ostrovsky, and Y. Rabani. "Approximation Algorithms for the Job Interval Selection Problem and Related Scheduling Problems." In *FOCS '01*.
- [CK-00] C. Chekuri and S. Khanna. "A PTAS for the Multiple Knapsack Problem." In *SODA '00*, 213–222. *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*.
- [DSS-96] A. Dan, D. Sitaram, and P. Shahabuddin. "Dynamic Batching Policies for an On-Demand Video Server." *ACM Multimedia Systems Journal*, **4**(3):112–121, 1996.
- [DF⁺99] A. Devpura, J. W. Fowler, M. W. Carlyle, and I. Perez. "Minimizing Total Weighted Tardiness on Single Batch Process Machine with Incompatible Job Families". *MS Thesis*, 1999.
- [DN-99] D. Dobson and G. Nambimadom. "The Batch Loading and Scheduling Problem." Working Paper QM 92-03, University of Rochester, Rochester, NY.
- [GJ-79] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*. Freeman, 1979.
- [MU-98] S. V. Mehta and R. Uzsoy. "Minimizing Total Tardiness on a Batch Processing Machine with Incompatible Jobs Types." *IIE Transactions*, **32**:165–175, 1998.
- [ST-01] H. Shachnai and T. Tamir. "On Two Class-Constrained Versions of the Multiple Knapsack Problem." *Algorithmica*, **29**:442–467, 2001.
- [S-99] F. C. R. Spieksma. "On the Approximability of an Interval Scheduling Problem." *Journal of Scheduling*, **2**:215–227, 1999.
- [U-95] R. Uzsoy. "Scheduling Batch Processing Machines with Incompatible Job Families." *Int. J. of Prod. Res.*, **33**:2685–2708, 1995.