Report

Single machine batch scheduling with release times

Author(s):
Gfeller, Beat

Publication Date:
2006

Permanent Link:

https://doi.org/10.3929/ethz-a-006780781 →

Single Machine Batch Scheduling with Release Times

Beat Gfeller* Leon Peeters* Birgitta Weber†
Peter Widmayer*

Institute of Theoretical Computer Science, ETH Zurich
Technical Report 514

April 4, 2006

Abstract

We consider the single-familiy scheduling problem with batching on a single machine, with a setup time and release times, under batch availability. We investigate the objective of minimizing the total flow time, in both the online and offline setting. For the online problem, we propose a 2-competitive algorithm for the case of identical processing times, and we prove a lower bound that comes close. With different processing times and job reordering, our lower bound shows that online algorithms are inevitably bad in the worst case. The offline problem is in P if jobs have to be processed in release order, and becomes NP-hard if reordering is allowed.

1 Introduction

The study in this paper is motivated by the real world problem of saving a log of actions in a high throughput environment. Many actions are to be carried out in rapid succession, and in case of a system failure the log can identify which actions have been carried out before the failure and which have not. Keeping such a log can be existential for a business, for example when logging the trading data in a stock brokerage company.

Logging takes place on disk and is carried out by a storage system that accepts write requests. When a process wants its data to be logged, it sends a log request with the log data to the storage system and waits for the acknowledgement that the writing of the log data has been completed. For the log requests that arrive over time at the storage system, there is only one decision the system is free to make: What subset of the requested, but not yet written log data should be written to disk in a single large write operation to make the whole system as efficient as possible? After the chosen large write operation is complete, the system instantaneously sends acknowledgements to all processes whose requests have been satisfied.

The difficulty in the above question comes from the fact that log data come in all sizes (number of bits or blocks to be stored), that writing several log data in a single shot is faster than writing each of them individually (due to the disk hardware constraints), and that a process requesting a write has to wait for the acknowledgement (of the completion of the large write operation) before it can continue. Based on an experimental evaluation of writing data to disk, we assume the writing time for a number of data blocks to be linear in that number, plus an additive constant (for the disk write setup time). Our objective

*Institute of Theoretical Computer Science, ETH Zurich, {gfeller,peetersl,widmayer}@inf.ethz.ch
†Department of Computer Science, University of Liverpool, weberb@csc.liv.ac.uk

is to minimize the sum over all requests of the times between the request's arrival and its acknowledgement. We ignore the details of a failure and its recovery here, and are not worrying about (the potential loss of) unsatisfied write requests.

1.1 Single Machine Scheduling with Batching

Viewing the storage system as a machine, the log requests as jobs, and the write operations as batches, this problem falls into the realm of scheduling with batching (Brucker et al., 1998, Brucker and Kovalyov, 1996, Chen et al., 2004, Cheng et al., 1996, Cheng and Kovalyov, 2001, Janiak et al., 2005, Kovalyov et al., 2004). More precisely, in the usual batch scheduling taxonomy (see the overview by Potts and Kovalyov, 2000) we deal with a *family scheduling problem* with batching on a single machine (with one family). The machine processes the jobs consecutively, since the log data are stored consecutively in time on the disk (as opposed to simultaneously), and each batch of jobs requires a constant (disk write) setup time. As all log requests in a single write are simultaneously acknowledged at the write completion time, the machine operates with *batch availability*, meaning that each job in the batch completes only when the full batch is completed.

In more formal terms, we model the storage system as a single machine, and the log requests as a set of jobs $J = \{1, \ldots, n\}$. The arrival times of the log requests at the storage system then correspond to job release times $r_j, j \in \{1, \ldots, n\}$. Further, each job $j \in J$ has a processing time p_j on the machine, representing the block size of the log request. The grouping of the log requests into write operations is modeled by the batching of the jobs into a partition $\sigma = \{\sigma_1, \ldots, \sigma_k\}$ of the jobs $\{1, \ldots, n\}$, where σ_u represents the jobs in the u-th batch, k is the total number of batches, and we refer to $|\sigma_u|$ as the *size* of batch u. Unless stated otherwise, we assume that the batch size is not limited. We denote the starting time of batch σ_u by T_u, with $r_j \leq T_u$ for all $j \in \sigma_u$. Starting at T_u, the batch requires the constant disk setup time s, and further a total processing time $\sum_{j \in \sigma_u} p_j$, for writing the logs on the disk. Thus, each batch σ_u requires a total *batch processing time* $P_u = s + \sum_{j \in \sigma_u} p_j$. The consecutive execution of batches on the single machine translates into $T_u + P_u \leq T_{u+1}$ for $u = 1, \ldots, k-1$. Because of batch availability, each job $j \in \sigma_u$ completes at time $C_j = T_u + P_u$, and takes a flow time $F_j = C_j - r_j$ to be processed. This job flow time basically consists of two components: first a *waiting time* $T_u - r_j \geq 0$ that the job waits before batch σ_u starts, followed by the batch processing time P_u. Finally, as mentioned above, our objective is to minimize the total job flow time $\mathcal{F} = \sum_{j=1}^{n} F_j$.

We refer to this scheduling problem as the BatchFlow problem. In the standard scheduling classification scheme, the BatchFlow problem is written as $1|r_j, s_f = s, F = 1|\sum F_j$, where the part $s_f = s, F = 1$ refers to the fact that the jobs belong to a single family with a fixed batch setup time (see Potts and Kovalyov, 2000). As special cases, we consider the problem variants with *identical processing times* $p_j = p$, and with a *fixed job sequence*, where jobs are to be processed in release order. In some cases, we constrain the flow time of each job by a *maximum flow time*, denoted by F_{max}.

1.2 Online Algorithms for a Single Machine with Batching

A large part of this paper considers the online version of the single machine scheduling problem with batching. Here, the jobs arrive over time, and any algorithm can base its batching decisions at a given time instant only on the jobs that have arrived so far. We study the online problem under the *non-preemptive clairvoyant* setting: No information about a job is known until it arrives, but once a job j has arrived, both its release time r_j (that is, arrival time) and processing time p_j are known. A batch that has started processing cannot be stopped before completion.

We consider *deterministic* online algorithms, and assume without loss of generality that the algorithms have a particular structure, as described in the following.

By definition, no new batch can be started as long as the machine is busy. Further, it only makes sense for an online algorithm to revise a decision when new information becomes available, that is, when a new job arrives. Therefore, we consider online algorithms that only take a decision at the completion time of a batch σ_u, or when a new job j arrives and the machine is idle. We refer to these two events as triggering events. In either case, the algorithm bases its decision on the jobs $\{1, \ldots, j\}$ that have arrived so far, and on the batches $\sigma_1, \ldots, \sigma_u$ created so far. Note that the set $\mathcal{P}$ of currently pending jobs can be deduced from this information.

In case of a triggering event, an online algorithm $\mathcal{A}$ takes the following two decisions. First, it tentatively chooses the next batch $\sigma_\mathcal{A}$ to be executed on the machine. However, it does not execute the batch $\sigma_\mathcal{A}$ immediately. Rather, the algorithm's second decision defines a *delay time* $\Delta_\mathcal{A}$ by which it delays the execution of $\sigma_\mathcal{A}$, and waits for a triggering event to occur in the meantime. If $\Delta_\mathcal{A}$ time has elapsed, and no triggering event has happened, then the algorithm starts the batch $\sigma_\mathcal{A}$ on the machine (by definition, the machine is idle in this case). If, however, a triggering event occurs during the delay time, then the algorithm newly chooses $\sigma_\mathcal{A}$ and $\Delta_\mathcal{A}$. Thus, an online algorithm $\mathcal{A}$ is completely specified by how it chooses $\sigma_\mathcal{A}$ and $\Delta_\mathcal{A}$.

1.3 Related Work

Since $\sum_{j \in J} r_j$ is a constant that we cannot influence, the offline version of our problem is equivalent to the problem $1|r_j, s_f = s, F = 1|\sum C_j$. The related problem $1|s_f = s, F = 1|\sum C_j$, without release times but with individual processing times, was first considered by Coffman et al. (1990). They solve this problem in $O(n \log n)$ time, first sorting the jobs by processing time, and then using an $O(n)$ dynamic programming solution. Albers and Brucker (1993) extend that solution to solve $1|s_f = s, F = 1|\sum w_j C_j$ for a fixed job sequence in $O(n)$ time, and show that the unrestricted problem $1|s_f = s, F = 1|\sum w_j C_j$ is unary NP-hard. Webster and Baker (1995) describe a dynamic program with running time $O(n^3)$ for the so-called *batch processing model*, where each batch has the same size-independent processing time, but the batch size is limited.

More recently, Cheng and Kovalyov (2001) describe complexity results for various related problems and objectives, also considering due dates, but not release times. They consider the *bounded* model, where the size of a batch[1] can be at most B, as well as the *unbounded* model.

The objective of minimizing the total completion time (weighted or unweighted) has been considered also in the so-called *burn-in* model (see Lee et al., 1992), where the processing time of a batch is equal to the maximum processing time among all jobs in the batch. For this model, Poon and Yu (2004) present two algorithms for $1|s_f = s, F = 1, B|\sum C_j$ with batch size bound B, with running times $O(n^{6B})$ and $n^{O(\sqrt{n})}$. Furthermore, Deng et al. (2004) consider $1|s_f = s, F = 1|\sum w_j C_j$ in the burn-in model with unbounded batch size. They show NP-hardness of that problem, and give a polynomial time approximation scheme.

Concerning the online setting, most previous work focuses on the burn-in model. An exception is Divakaran and Saks (2001), who consider the problem $1|r_j, s_f|\max F_j$ with sequence-independent setup times and several job families under job availability (i.e. the processing of each job completes as soon as its processing time has elapsed). They present an $O(1)$-competitive online algorithm for that problem. For the burn-in model, Chen et al.

[1]This bound is also called *capacity* by some authors.

(2004) consider the problem $1|r_j|\sum w_jC_j$, and present a 10/3-competitive online algorithm for unbounded batch size, as well as a $4+\epsilon$-competitive online algorithm for bounded batch size.

The online problem $1|r_j|C_{\max}$ of minimizing the makespan in the burn-in model has been considered in several studies. Independently, Deng et al. (2003) and Zhang et al. (2001) gave a $(\sqrt{5}+1)/2$ lower bound for the competitive ratio, and both gave the same online algorithm for the unbounded batch size model which matches the lower bound. In Poon and Yu (2005a), they present a different online algorithm with the same competitiveness, and describe a parameterized online algorithm which contains their own and the previous solution as special cases. The same authors give a class of 2-competitive online algorithms for bounded batch size in Poon and Yu (2005b), and a 7/4-competitive algorithm for batch size limit $B = 2$.

1.4 Contribution of the Paper

To our knowledge, we are the first to consider release times with the objective of minimizing the total flow time $\sum F_j$ under batch availability. In this setting, we study the offline problem to some degree, but are mainly interested in solutions to its online version.

For the online BatchFlow problem, we introduce the Greedy online algorithm. For the special case of identical processing times p, we show that Greedy is strictly 2-competitive, using the fact that its makespan is optimal up to an additive constant. Moreover, we present two lower bounds, $1+\frac{1}{1+\frac{\max(p,s)}{\min(p,s)}}$ and $1+\frac{1}{1+2\frac{p}{s}}$ for this problem variant, and hence show that Greedy is not far from optimal for this variant.

For the general online BatchFlow problem, we then give an $\frac{n}{2}-\epsilon$ lower bound for the competitive ratio, and show that any online algorithm which avoids unnecessary idle time, including Greedy, is strictly n-competitive, which matches the order of the lower bound. Furthermore, we show that in general, no online algorithm can enforce an upper bound on the flow time.

We show that the general offline problem $1|r_j, s_f = s, F = 1|\sum F_j$ is NP-hard, which contrasts the $O(n \log n)$ algorithm by Coffman et al. (1990) for the case without release times. For the fixed job sequence case, we present an $O(n^5)$ dynamic program, which complements the previously mentioned $O(n)$ dynamic program for the same problem without release times by Albers and Brucker (1993).

2 The online BatchFlow problem

We start our investigations with the online BatchFlow problem. For a given problem instance I, let $\mathcal{F}_{\text{Opt}}$ be the total flow time of an optimal solution, and $\mathcal{F}_{\mathcal{A}}$ the total flow time of the solution obtained from some online algorithm $\mathcal{A}$ for the same problem. We are interested in the *competitive ratio* $\frac{\mathcal{F}_{\mathcal{A}}}{\mathcal{F}_{\text{Opt}}}$ of an online algorithm $\mathcal{A}$. The online algorithm $\mathcal{A}$ is *c-competitive* if there is a constant α such that for all finite instances I, $\mathcal{F}_{\mathcal{A}}(I) \leq c \cdot \mathcal{F}_{\text{Opt}}(I) + \alpha$. When this condition holds also for $\alpha = 0$, we say that $\mathcal{A}$ is *strictly c-competitive*.

We first analyze the online BatchFlow problem for jobs with identical processing times $p_i = p$. For this case, we present a 2-competitive greedy algorithm in Section 2.1, and derive two lower bounds in terms of p and s for any online algorithm in Section 2.2. Next, Section 2.3 discusses bounds for any online algorithm for the case of general processing times. In these investigations, we ignore the maximum flow time constraint for reasons explained in Section 2.4.

2.1 The GREEDY batching algorithm for identical processing times

In this section, we consider the restricted case where all jobs have identical processing times $p_i = p$. This case is relevant in applications such as ours, where records of fixed length are to be logged. Note that the reordering of jobs with identical processing times is never beneficial, so it is irrelevant whether the fixed job sequence restriction is present or not.

We now define the GREEDY algorithm, which always starts a batch consisting of all currently pending jobs as soon as the machine becomes idle. Formally, GREEDY chooses $\Delta_{\mathcal{A}} = 0$, and sets $\sigma_{\mathcal{A}}$ equal to the set of currently pending jobs $\mathcal{P}$. First, we focus on the makespan of GREEDY. The following theorem shows that, if GREEDY needs time t to finish a set of batches $\{\sigma_1, \ldots, \sigma_u\}$, then no other algorithm can complete the same jobs before time $t - s$. Thus, GREEDY is 1-competitive for minimizing the makespan, with an additive constant $\alpha = s$.

Theorem 2. *For a given problem instance of the online* BATCHFLOW *problem with identical processing times, let $\sigma = \sigma_1, \ldots, \sigma_k$ with batch starting times $T_1, \ldots, T_k$ be the* GREEDY *solution, and let $\sigma' = \sigma'_1, \ldots, \sigma'_m$ with $T'_1, \ldots, T'_m$ be some other solution* ANY *for the same instance. For any batch $\sigma_u \in \sigma$ completing at time t, it holds that any batch $\sigma'_v \in \sigma'$ satisfying the condition*

$$\sum_{i=1}^{v} |\sigma'_i| \geq \sum_{i=1}^{u} |\sigma_i| \tag{1}$$

completes at time $t' \geq t - s$.

Proof. For a given batch $\sigma_u \in \sigma$ consider the first batch $\sigma'_v \in \sigma'$ for which (1) holds. Such a batch exists because $\sum_{i=1}^{m} |\sigma'_i| = n$ and of course $\sum_{i=1}^{u} |\sigma_i| \leq n$. We assume that the GREEDY batches $\sigma_1, \ldots, \sigma_u$ are executed without any idle time in between. Indeed, if such an idle time occurs, GREEDY must have processed all jobs which have arrived so far, and the idle time ends exactly at the release time of the next job. Of course, ANY cannot start processing this job earlier than GREEDY does. Hence, ignoring all jobs before such an idle time can only affect the comparison in favor of ANY.

First, we consider the case $u \geq v + 1$, where GREEDY uses at least one batch more than ANY. We require the following lemma, which is illustrated in Figure 1.

Lemma 3. *Consider a sequence $\sigma_a, \ldots, \sigma_b$ of u* GREEDY *batches, and a sequence $\sigma'_c, \ldots, \sigma'_d$ of v* ANY *batches, such that the* ANY *sequence contains all the jobs in the* GREEDY *sequence, and possibly additional jobs. If $u \geq v+1$, then there exists a batch σ'_* among* ANY*'s batches that contains both at least one entire* GREEDY *batch, and at least one following job J_* from the next* GREEDY *batch.*

Proof. Instead of proving the lemma directly, we prove the following equivalent statement: *Assuming that no batch among $\sigma'_c, \ldots, \sigma'_d$ fully contains a* GREEDY *batch plus a following job, it holds that $u \leq v$.*

We show that claim by induction over u. For $u = 1$ the claim is trivially true. Suppose that the claim holds for $1, \ldots, u-1$. By our assumption, the first ANY batch σ'_c can at most contain σ_a entirely, but no following jobs. Thus, the jobs in $\sigma_{a+1}, \ldots, \sigma_b$ must be covered by $\sigma'_{c+1}, \ldots, \sigma'_d$. These two sequences have lengths $u - 1$ and $v - 1$, respectively, so by the induction hypothesis, we have $1 + u - 1 \leq 1 + v - 1$, thus $u \leq v$. □

Apply Lemma 3 to $\sigma_1, \ldots, \sigma_u$ and $\sigma'_1, \ldots, \sigma'_v$, and let σ'_* be the *last* ANY batch containing a full GREEDY batch followed by at least one job. Choose σ_* such that it is the last full GREEDY batch in σ'_* that is followed by some job J_* in σ'_*. When J_* arrives, GREEDY is already processing batch σ_* (or has just finished), because otherwise J_* would be part of

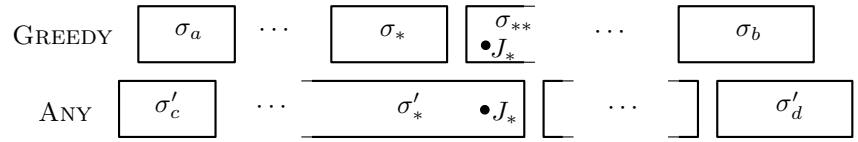

Figure 1: Illustration of Lemma 3.

σ_*. On the other hand, ANY cannot start processing batch σ'_* before J_* arrives. So, it must hold that $T_{\sigma_*} \leq T_{\sigma'_*}$. Let σ_{**} be the GREEDY batch following σ_*. Note that Lemma 3 (in contraposition) can be applied also to the sequences $\sigma_{**}, \ldots, \sigma_u$ and $\sigma'_*, \ldots, \sigma'_v$. Thus, since in these two sequences no ANY batch contains an entire GREEDY batch followed by another job, we have $|\{\sigma_{**}, \ldots, \sigma_u\}| \leq |\{\sigma'_*, \ldots, \sigma'_v\}|$. Defining z as the number of batches in $\{\sigma_*, \ldots, \sigma_u\}$, and z' as the number of batches in $\{\sigma'_*, \ldots, \sigma'_v\}$, it holds $z \leq z' + 1$. Putting all of the above together, we obtain:

$$t - t' \leq T_{\sigma_*} + p \cdot (|\sigma_*| + \ldots + |\sigma_u|) + zs - T_{\sigma'_*} - p \cdot (|\sigma'_*| + \ldots + |\sigma'_v|) - z's \leq s$$

Finally, we consider the remaining case $u \leq v$. Since GREEDY has no idle time, it completes σ_u exactly at time $t = p \cdot \sum_{i=1}^{u} |\sigma_i| + us$. ANY finishes σ'_v at time $t' \geq p \cdot \sum_{i=1}^{v} |\sigma'_i| + vs$ or later. So, using (1), we obtain that $t' - t \geq (v - u)s$, proving the theorem for any $u \leq v + 1$, and for $u \leq v$ in particular. □

From this theorem, we obtain the following lemma.

Lemma 4. *In the online* BATCHFLOW *problem with identical processing times, consider any batch σ_u of the* GREEDY *solution, with starting time T_u. Let σ' be the first batch of the optimal solution* OPT *that contains some job in σ_u. The earliest time that* OPT *can finish processing the m jobs in $\sigma_u \cap \sigma'$ is $T_u + mp$.*

Proof. Observe that, if we deleted all jobs in $\sigma_u \backslash \sigma'$ from the problem instance, then GREEDY would start processing exactly the m jobs in $\sigma_u \cap \sigma'$ in one batch at time T_u, and finish at $T_u + mp + s$. Now, if OPT were to finish these m jobs before $T_u + mp$, then there would exist a solution with makespan more than s smaller than GREEDY's makespan. This is a contradiction to Theorem 2. □

Note that the proofs of Theorem 2 and Lemma 4 can easily be adapted to incorporate non-identical processing times. We proved them for identical processing times here, since they serve as ingredients for the main theorem below, which only applies to identical processing times.

Now follows an observation concerning the optimal offline solution for a special case, which we need afterwards to compare online algorithms against the best possible solution.

Observation 5. *The total job flow time for optimally processing n jobs $1, \ldots, n$ with identical processing times $p_j = p$ and with identical release times is at least*

$$\mathcal{F}_n \geq \frac{1}{2}pn^2 + sn.$$

Proof. Assume w.l.o.g. that all $r_j = 0$. Consider the first job: This job will have completion time at least $s + p$. The second job will finish no earlier than $s + 2p$, which can be achieved if the first two jobs are batched together. Generally, the ith job can finish no earlier than $s + ip$, which would be achieved by batching the first i jobs together. This shows that $\mathcal{F}_n \geq \sum_{i=1}^{n}(s + ip) = \frac{1}{2}pn(n + 1) + sn \geq \frac{1}{2}pn^2 + sn$. □

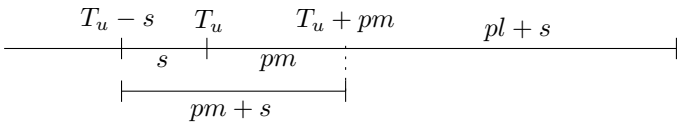

Figure 2: Important time instants in GREEDY's competitiveness proof.

Theorem 6. *The* GREEDY *algorithm is strictly 2-competitive for the online* BATCHFLOW *problem with identical processing times.*

Proof. Figure 2 shows all the relevant time instants for the proof. As in Lemma 4, consider any batch σ_u of the GREEDY solution, with starting time T_u, and let σ' be the first batch of the optimal solution OPT that contains some jobs in σ_u. Below, we compare the total accumulated flow time before and after time T_u for the jobs in σ_u, for both GREEDY and OPT.

Lemma 4 implies that no job in σ_u can complete before T_u in OPT. Thus, up until time T_u, the jobs in σ_u have accumulated a total flow time of $\mathcal{F}^{\leq T_u}(\sigma_u) := \sum_{j \in \sigma_u}(T_u - r_j)$ in both GREEDY and OPT.

Let $\mathcal{F}^{\geq T_u}_{\text{GREEDY}}(\sigma_u)$ denote the total flow time for the jobs in σ_u after time T_u in the GREEDY solution, and $\mathcal{F}^{\geq T_u}_{\text{OPT}}(\sigma_u)$ the same quantity for the OPT solution. Further, we let $\mathcal{F}_{\text{OPT}}(\sigma_u) = \mathcal{F}^{\geq T_u}_{\text{OPT}}(\sigma_u) + \mathcal{F}^{\leq T_u}(\sigma_u)$ and $\mathcal{F}_{\text{GREEDY}}(\sigma_u) = \mathcal{F}^{\geq T_u}_{\text{GREEDY}}(\sigma_u) + \mathcal{F}^{\leq T_u}(\sigma_u)$ be the total flow time for the jobs in σ_u in OPT and GREEDY, respectively.

Now, if σ' starts at $T_u - s$ or earlier, then all jobs in σ' must arrive at $T_u - s$ or earlier. Therefore, up until time T_u, the m jobs in $\sigma_u \cap \sigma'$ already yield an accumulated total flow time $\mathcal{F}^{\leq T_u}(\sigma_u) \geq ms$ in this case. After time T_u, the GREEDY solution further accumulates a total flow time $\mathcal{F}^{\geq T_u}_{\text{GREEDY}}(\sigma_u) = P_u = l(lp+s)$ for the l jobs in σ_u. As σ' finishes at or after T_u, and all jobs in σ_u must have arrived by T_u, the total flow time of OPT for the jobs in σ_u after time T_u is

$$\mathcal{F}^{\geq T_u}_{\text{OPT}}(\sigma_u) \geq \overbrace{m(pm)}^{\text{Lemma 4}} + \overbrace{(l-m)(pm)}^{\text{wait for } \sigma' \text{ to complete}} + \overbrace{\frac{1}{2}p(l-m)^2 + (l-m)s}^{\text{Observation 5}} = \frac{1}{2}(pl^2+pm^2)+s(l-m). \tag{2}$$

Therefore, we have

$$\begin{aligned} 2 \cdot \mathcal{F}_{\text{OPT}}(\sigma_u) &\geq pl^2 + pm^2 + 2s(l-m) + 2\mathcal{F}^{\leq T_u}(\sigma_u) \geq pl^2 + s(l-m) + 2\mathcal{F}^{\leq T_u}(\sigma_u) \\ &\geq pl^2 + sl + \mathcal{F}^{\leq T_u}(\sigma_u) = \mathcal{F}_{\text{GREEDY}}(\sigma_u). \end{aligned} \tag{3}$$

Next, we consider the case in which σ' starts after $T_u - s$, say at starting time $T_u - s + \tau$, for $\tau > 0$. We still have $\mathcal{F}^{\geq T_u}_{\text{GREEDY}}(\sigma_u) = l(lp+s)$ for GREEDY. In this case, however, we obtain $\mathcal{F}^{\leq T_u}(\sigma_u) \geq m(s-\tau)$, and further an additive term $l\tau$ in the bound (2) for $\mathcal{F}^{\geq T_u}_{\text{OPT}}(\sigma_u)$. The additive term $l\tau$ for $\mathcal{F}^{\geq T_u}_{\text{OPT}}(\sigma_u)$ cancels out against the extra term $-m\tau$ for $\mathcal{F}^{\leq T_u}(\sigma_u)$ in the inequalities (3), so the bound (3) also applies in this case.

Since $\frac{\mathcal{F}_{\text{GREEDY}}(\sigma_u)}{\mathcal{F}_{\text{OPT}}(\sigma_u)} \leq 2$ holds for any batch σ_u of the GREEDY solution, the theorem follows. □

2.2 Lower bounds for identical processing times

Below, we derive two lower bounds for any algorithm for the online BATCHFLOW problem, again with identical processing times. These bounds show that no online algorithm can be much better than GREEDY for this setting.

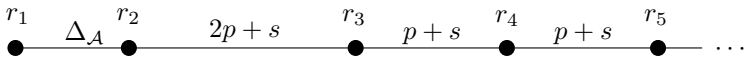

Figure 3: The lower bound construction.

Theorem 7. *No online algorithm for the online* BatchFlow *problem with identical processing times can have a competitive ratio lower than*

$$1 + \frac{1}{1 + \frac{\max(p,s)}{\min(p,s)}}.$$

Proof. Let $\mathcal{A}$ be any online algorithm with finite delay time $\Delta_{\mathcal{A}}$ for $\mathcal{P} = \{1\}$. The adversary chooses release times $r_1 = 0$, $r_2 = \Delta_{\mathcal{A}}$, and $r_j = \Delta_{\mathcal{A}} + p(j-1) + s(j-2)$ for $j \in \{3, \ldots, n\}$, as depicted in Figure 3. Observe that an offline solution can avoid any waiting time for $n-2$ jobs: If job 1 and job 2 are processed together, the first batch has finished just when job 3 arrives, so if job 3 is processed immediately, it will be finished just when job 4 arrives, and so on until job n. Thus,

$$\mathcal{F}_{\text{OPT}} \leq \overbrace{\Delta_{\mathcal{A}}}^{\text{job 1 waits}} + 2(2p+s) + (n-2)\cdot(p+s) = n(p+s) + \Delta_{\mathcal{A}} + 2p.$$

For bounding the flow time of $\mathcal{A}$'s solution, we examine for each job j the earliest possible completion time that $\mathcal{A}$ can achieve.

By construction of the example, $\mathcal{A}$ cannot batch job 2 together with job 1, and starts processing job 1 at time $\Delta_{\mathcal{A}}$, which completes at $C_1 = \Delta_{\mathcal{A}} + p + s$. Hence, job 2 cannot start processing before C_1, and thus $C_2 \geq \Delta_{\mathcal{A}} + 2p + 2s$. By induction, we show that for each $j \in \{3, \ldots, n\}$, it holds $C_j \geq \Delta_{\mathcal{A}} + pj + s(j-1) + \min(p,s)$.
$\mathbf{j = 3}$: As batching job 3 together with later jobs will only increase job 3's completion time, the earliest possible completion time C_3 is either achieved by batching job 3 with job 2, or by processing job 3 separately. The two possibilities yield completion times

$$C_3' = \Delta_{\mathcal{A}} + 2p + s + \overbrace{2p+s}^{\text{processing job 2 and job 3}} = \Delta_{\mathcal{A}} + 4p + 2s$$

and

$$C_3'' = \Delta_{\mathcal{A}} + 2p + 2s + p + s = \Delta_{\mathcal{A}} + 3p + 3s,$$

respectively. We have $C_3 \geq \Delta_{\mathcal{A}} + 3p + 2s + \min(p,s)$.
$\mathbf{j-1 \to j}$: Note that batching more than two jobs would result in an idle time of more than $p+s$, which is certainly not fastest possible. So, the fastest possible way to process job j is to either batch it with job $j-1$ or to process it separately. If job j is batched with job $j-1$, then

$$C_j \geq r_j + 2p + s = \Delta_{\mathcal{A}} + p(j+1) + s(j-1).$$

If job j is processed separately,

$$C_j \geq C_{j-1} + p + s \geq \Delta_{\mathcal{A}} + p(j-1) + s(j-2) + \min(p,s) + p + s = \Delta_{\mathcal{A}} + pj + s(j-1) + \min(p,s).$$

Again, we see that $C_j \geq \Delta_{\mathcal{A}} + pj + s(j-1) + \min(p,s)$.

Adding $\sum_{j=1}^{n} F_j = \sum_{j=1}^{n} C_j - r_j$, we get

$$\mathcal{F}_{\mathcal{A}} \geq \overbrace{\Delta_{\mathcal{A}} + p + s}^{\text{job 1}} + \overbrace{2p+2s}^{\text{job 2}} + \sum_{j=3}^{n} (p + s + \min(p,s)) = \Delta_{\mathcal{A}} + np + p + ns + s + (n-2)\min(p,s)$$

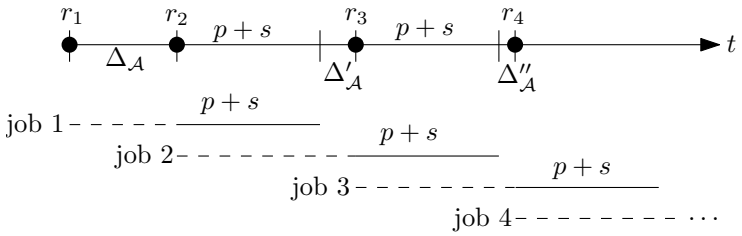

Figure 4: Forcing $\mathcal{A}$ to process all jobs separately. Waiting time is shown as dashed lines, batch processing time as solid lines.

The competitive ratio can now be bounded as

$$\frac{\mathcal{F}_{\mathcal{A}}}{\mathcal{F}_{\text{OPT}}} \geq \frac{\Delta_{\mathcal{A}} + np + p + ns + s + (n-2)\min(p,s)}{n(p+s) + \Delta_{\mathcal{A}} + 2p}$$

$$\to \frac{p+s+\min(p,s)}{p+s} = 1 + \frac{1}{1+\frac{\max(p,s)}{\min(p,s)}} \quad \text{for } n \to \infty.$$

□

Using a similar construction, one can show the following lower bound.

Theorem 8. *No online algorithm for the online* BatchFlow *problem with identical processing times can have a competitive ratio lower than*

$$1 + \frac{1}{1+2\frac{p}{s}}.$$

Proof. The following adversary can force every online algorithm $\mathcal{A}$ to process all jobs separately: Release job 1 and wait for $\Delta_{\mathcal{A}}$ until $\mathcal{A}$ processes it, then immediately release job 2, wait for $\Delta'_{\mathcal{A}}$ until $\mathcal{A}$ processes it, and so on. Note that this causes every job except the first one to wait at least $p+s$ (see Figure 4). Furthermore, the intervals between the job releases are all at least $p+s$ (and exactly $p+s$ if $\mathcal{A}$ processes each job as soon as the previous batch finishes, i.e. all $\Delta'_{\mathcal{A}} = \Delta''_{\mathcal{A}} = \ldots = 0$), except for the first interval $\Delta_{\mathcal{A}}$, which can be smaller. The costs of $\mathcal{A}$'s solution thus are the batch processing time plus at least $p+s$ waiting time for $n-1$ jobs, giving

$$\mathcal{F}_{\mathcal{A}} \geq \overbrace{n(p+s)}^{\text{processing}} + \overbrace{\Delta_{\mathcal{A}} + (n-1)(p+s)}^{\text{waiting}} = (2n-1)(p+s) + \Delta_{\mathcal{A}}.$$

For bounding the offline costs, we consider the solution of batching job 1 and job 2 together, and processing all other jobs separately. Since each interval after job 2 is at least $p+s$, and processing job 1 and job 2 together takes $2p+s$ time, job 3 needs to wait at most p. Processing job 3 will hence complete no later than $r_4 + p$, so job 4 has to wait at most p, and so on. So, we can bound the optimal offline costs as

$$\mathcal{F}_{\text{OPT}} \leq \Delta_{\mathcal{A}} + \overbrace{2(2p+s)}^{\text{process jobs } 1,2} + \overbrace{(n-2)p}^{\text{waiting of jobs } 3,\ldots,n} + \overbrace{(n-2)(p+s)}^{\text{process jobs } 3,\ldots,n} = \Delta_{\mathcal{A}} + 2np + ns.$$

Thus, the competitive ratio is bounded by

$$\frac{\mathcal{F}_{\mathcal{A}}}{\mathcal{F}_{\text{OPT}}} \geq \frac{2p + 2s - \frac{p+s-\Delta_{\mathcal{A}}}{n}}{2p + s + \frac{\Delta_{\mathcal{A}}}{n}} \to 1 + \frac{1}{1+\frac{2p}{s}} \quad \text{for } n \to \infty.$$

□

2.3 Bounds on the competitive ratio with job reordering

The following theorem shows that no online algorithm can have a good worst case performance for the online BatchFlow problem with general processing times, if the job sequence is not fixed.

Theorem 9. *For the online* BatchFlow *problem, any online algorithm has competitive ratio at least* $\frac{n}{2} - \epsilon$ *for any* $\epsilon > 0$.

Proof. Let $\mathcal{A}$ be any online algorithm. Consider an instance of n jobs, where the first job 1 has processing time p, and all other jobs $2, \ldots, n$ have processing time 1, and arrive immediately after $\mathcal{A}$ starts processing job 1 (after having delayed for $\Delta_{\mathcal{A}}$ time units). As each of the jobs $2, \ldots, n$ has to wait for job 1 to finish, the total flow time for $\mathcal{A}$ is

$$\mathcal{F}_{\mathcal{A}} \geq \Delta_{\mathcal{A}} + n(p+s) + \frac{1}{2}(n-1)^2 + (n-1)s,$$

where we used the lower bound from Observation 5 for optimally processing $(n-1)$ jobs of equal processing time arriving at the same time.

For Opt, consider the solution that first processes jobs $2, \ldots, n$ in one batch, and after that processes job 1:

$$\mathcal{F}_{\text{Opt}} \leq \Delta_{\mathcal{A}} + n((n-1)+s) + p + s.$$

We assume in the following that $\Delta_{\mathcal{A}} \leq p + s$; if $\Delta_{\mathcal{A}} > p + s$, then our bound for $\mathcal{F}_{\mathcal{A}}$ increases, but we can decrease the bound for $\mathcal{F}_{\text{Opt}}$ because Opt can complete job 1 even before the other jobs arrive, and then process all other jobs in one batch. We thus have

$$\mathcal{F}_{\mathcal{A}} \geq np + \frac{1}{2}n^2 + 2ns - n - s + \frac{1}{2} \quad \text{and} \quad \mathcal{F}_{\text{Opt}} \leq 2p + 2s + n^2 - n + ns.$$

It is easily verified that

$$\left(\frac{n}{2} - \epsilon\right) \cdot \mathcal{F}_{\text{Opt}} \leq \mathcal{F}_{\mathcal{A}} \quad \text{if we choose} \quad p \geq \frac{1}{4\epsilon}\left(n^3 + n^2 s + 2n + 2s + 2\epsilon n\right).$$

□

Theorem 9 shows that for the general setting, there is no online algorithm with a sublinear competitive ratio. However, we will see in the following that all so-called non-waiting algorithms, a class to which the Greedy algorithm belongs, are strictly n-competitive, i.e., are at most a factor 2 away from the lower bound. We call an online algorithm *non-waiting* if it never produces a solution in which there is idle time while some jobs are pending.

Theorem 10. *Any non-waiting online algorithm for the online* BatchFlow *problem is strictly* n*-competitive.*

Proof. Let $\mathcal{A}$ be any non-waiting online algorithm. Consider any job i, and let σ_u be the batch which contains job i. Furthermore, let J' be the set of all jobs not contained in σ_u. The flow time of job i is $F_i = C_i - r_i = T_u + P_u - r_i$. The longest possible interval during which σ_u needs to wait (i.e. the machine is busy) in a non-waiting algorithm's solution is $s|J'| + \sum_{j \in J'} p_j$. So, for a non-waiting algorithm, $T_u - r_i \leq s(n-1) + \sum_{j \in J'} p_j$. Hence,

$$F_i \leq P_u + s(n-1) + \sum_{j \in J'} p_j \leq sn + \sum_{j=1}^{n} p_j.$$

Thus, the total flow time for $\mathcal{A}$'s solution is

$$\mathcal{F}_{\mathcal{A}} = \sum_{i=1}^{n} F_i \leq n^2 s + n \cdot \sum_{j=1}^{n} p_j.$$

We now turn to the optimal solution Opt. Clearly, each job i has flow time $F_i' \geq p_i + s$, as the batch processing time is inevitable. Thus, the flow time of Opt is at least

$$\mathcal{F}_{\text{Opt}} = \sum_{i=1}^{n} F_i' \geq ns + \sum_{j=1}^{n} p_j.$$

Comparing the total flow times $\mathcal{F}_{\mathcal{A}}$ and $\mathcal{F}_{\text{Opt}}$ completes the proof. □

Observe that this upper bound proof does not make use of the fact that the reordering of jobs is allowed. Thus, adding a fixed job sequence constraint does not affect the validity of Theorem 10. Note that this is not true for Theorem 9.

We remark that the online non-preemptive scheduling problem with release times is a special case of the online BatchFlow problem. Thus, the $\Theta(n)$ upper bound for the former problem, mentioned by Epstein and van Stee (2003), is implied by our Theorem 10.

2.4 Maximum flow time constraint

Besides minimizing the total flow time, it may be desirable in many applications to have a guaranteed limit for the maximum flow time of any job, as this ensures that no job has to wait for an unbounded time. However, online algorithms have severe difficulties in dealing with such a maximum flow time constraint, as the following theorem describes more formally.

Theorem 11. *Consider the* BatchFlow *problem with a maximum flow time constraint. For any F, there is an $F_{max} \geq F$ such that, for any online algorithm $\mathcal{A}$, there exists an instance for which $\mathcal{A}$ produces an infeasible solution under the maximum flow time constraint F_{max}, although a feasible offline solution exists under F_{max}.*

The proof is based on two types of instances. In one instance type, feasibility is only possible by processing the first job as soon as it arrives, whereas in the other type, the first job must be batched together with later jobs in order to obtain a feasible solution. Whichever decision the online algorithm takes can be used by an adversary to make it fail.

Proof. All jobs used for the proof have processing time $p_i = 1$. Choose any $i \in \mathbb{N}^+$ such that $F_{\max} := i + s \geq F$. Let $\mathcal{A}$ be any online algorithm. Consider an instance where job 1 arrives at time $r_1 = 0$. We distinguish two cases:
Case A: $\mathcal{A}$ waits some time $\Delta_{\mathcal{A}}$ before starting to process job 1.
Let $n = i + 1$. All jobs $2, \ldots, n$ arrive at time $1 + s$. The fastest way of processing these i jobs is bundling them into one batch, as shown in Figure 5, which takes $i + s = (n - 1) + s$ time.

Hence, $\mathcal{A}$ cannot finish processing all these jobs fast enough, because the earliest time it can finish all is $\Delta_{\mathcal{A}} + 1 + s + (n - 1) + s = \Delta + n + 2s$, but the flow time of the last job would then be $\Delta_{\mathcal{A}} + n + 2s - (1 + s) = \Delta_{\mathcal{A}} + (n - 1) + s > F_{\max} = (n - 1) + s$. However, there is a feasible offline solution, as shown in Figure 6: if job 1 is processed immediately after arrival, it finishes at $1 + s$, and all other jobs can start processing at $1 + s$, and hence finish at $1 + s + (n - 1) + s = n + 2s$.
Case B: $\mathcal{A}$ starts processing job 1 immediately.
Let $n = i + 2$. Job 2 arrives at $r_2 = r_1 + \epsilon$. All other jobs $3, \ldots, n$ arrive $2 + s$ later. There

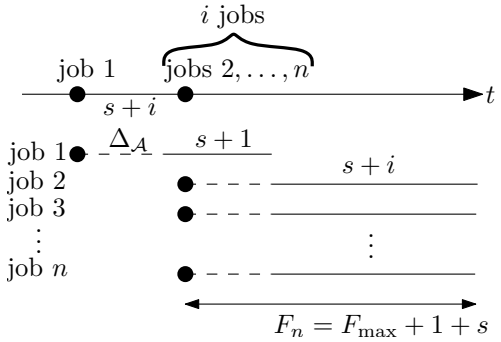

Figure 5: The fastest possible solution of $\mathcal{A}$ in Case A.

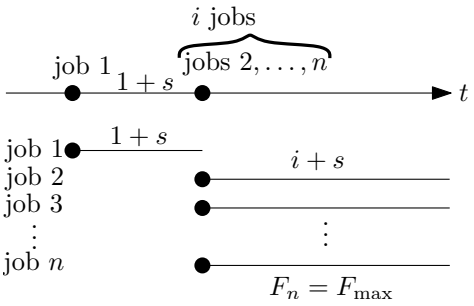

Figure 6: A feasible solution in Case A.

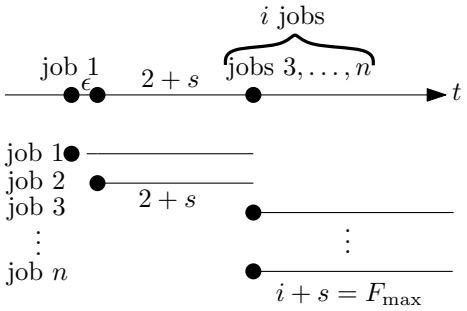

Figure 7: A feasible solution in Case B.

is a feasible offline solution for this instance, described in the following. Bundle job 1 and job 2 into one batch, which finishes at $\epsilon+2+s$. Hence, all other jobs $3, \ldots, n$ can be bundled into a batch which starts at $2+s+\epsilon$, and completes at $2+s+\epsilon+(n-2)+s = n+2s+\epsilon$, so the flow time of jobs $3, \ldots, n$ is $n+2s-(2+s) = (n-2)+s = i+s = F_{\max}$. This solution is shown in Figure 7. Observe that if jobs $3, \ldots, n$ are processed in a batch immediately as they arrive, their flow time is equal to $F_{\max}$. However, as $\mathcal{A}$ processes job 1 separately, it cannot finish processing job 2 until jobs $3, \ldots, n$ arrive. Thus, $\mathcal{A}$ cannot avoid that job n has a flow time larger than $F_{\max}$, violating the constraint. □

2.5 A lower bound for any constant k for a fixed job sequence (and $p_i \in \mathbb{Q}^+$)

In this section, we consider the online BatchFlow problem for a fixed job sequence, but with arbitrary rational processing times. We show that no online algorithm can have a constant competitive ratio.

Theorem 12. *There exists no online algorithm with a constant competitive ratio for the online* BatchFlow *problem with $p_i \in \mathbb{Q}^+$ and a fixed job sequence.*

The proof is by contradiction and consists of two main parts. First, it is shown that for each online algorithm with a constant competitive ratio k, and any $\gamma \in \mathbb{Q}$, there exists a job sequence which $\mathcal{A}$ finishes $\geq \gamma$ later than the fastest solution. Second, it is proven that this delay leads to a competitive ratio of at least $\frac{\gamma}{1+s}$ for $\mathcal{A}$ if many small jobs are added to the end of this job sequence.

Proof. By contradiction, assume there is an online algorithm $\mathcal{A}$ with constant competitive ratio k. First we introduce a construction we call "bursts". A "burst" consists of r jobs with processing time 1 arriving at the same time, immediately followed by one large job

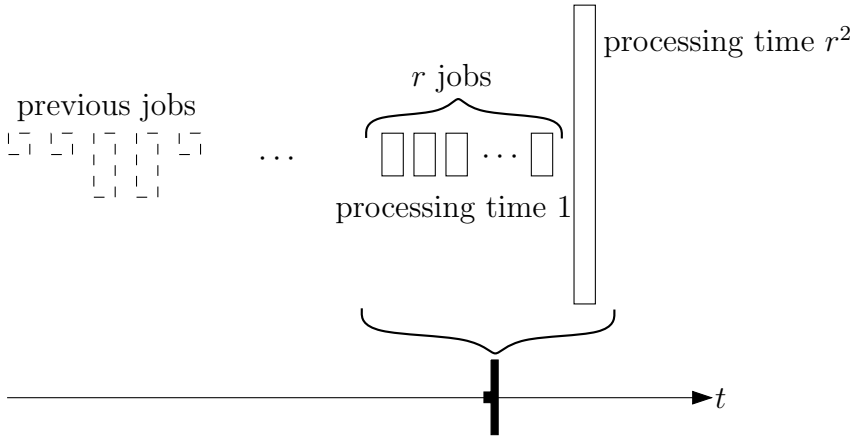

Figure 8: The structure of a burst. The small burst symbol drawn on the time axis is used in later figures.

with processing time r^2. Before the actual burst, there may be an arbitrary sequence of jobs, as shown in Figure 8. The reason for introducing the concept "burst" is that among the possible solutions for processing it, the fastest solution has high costs, whereas much cheaper solutions exist, but take more time (i.e. have a longer makespan). As $\mathcal{A}$ is by assumption k-competitive, it cannot choose the fastest solution if there is a solution which is k times cheaper. Thus, a burst causes $\mathcal{A}$ to have some delay compared to the fastest solution. This informal idea is explained formally in the following.

Throughout the proof, we assume that a burst is always placed at a point in time where $\mathcal{A}$ has already started the last batch of the previous jobs. This implies that $\mathcal{A}$ never bundles jobs of the burst together with previous jobs.

For such a burst, we consider two different solutions: If all jobs of the burst are processed in one batch, the incurred costs are

$$\mathcal{F}_{together} = (r+1)(r+r^2+s) = r^2 + r^3 + rs + r + r^2 + s.$$

On the other hand, if all small jobs of the burst are processed in one batch, and the large job separately, the incurred costs are

$$\mathcal{F}_{split} = r(r+s) + r + s + r^2 + s = r^2 + r + rs + 2s + r^2.$$

Let $\mathcal{F}_{\mathcal{A}}^{prev}$ be the costs of $\mathcal{A}$ for processing all previous jobs, including any waiting times for the jobs in the burst up to the point in time where all previous jobs have been processed. If $\mathcal{A}$ decided to bundle all jobs of the burst together into one batch, its total costs were

$$\mathcal{F}_{\mathcal{A}}^{\text{FAST}} = \mathcal{F}_{\mathcal{A}}^{prev} + \mathcal{F}_{together} = \mathcal{F}_{\mathcal{A}}^{prev} + (r+1)(r+r^2+s).$$

We will call this solution Π_{FAST}.
As for solutions with lower cost, consider a solution Π_{SLOW} which batches all previous jobs exactly as $\mathcal{A}$ does, but splits the burst into two batches, the first one containing all small jobs, and the second one containing only the large job. We have

$$\mathcal{F}_{\mathcal{A}}^{\text{SLOW}} = \mathcal{F}_{\mathcal{A}}^{prev} + \mathcal{F}_{split} = \mathcal{F}_{\mathcal{A}}^{prev} + r(r+s) + r + s + r^2 + s.$$

If the inequality

$$\frac{\mathcal{F}_{\mathcal{A}}^{\text{FAST}}}{\mathcal{F}_{\mathcal{A}}^{\text{SLOW}}} > k$$

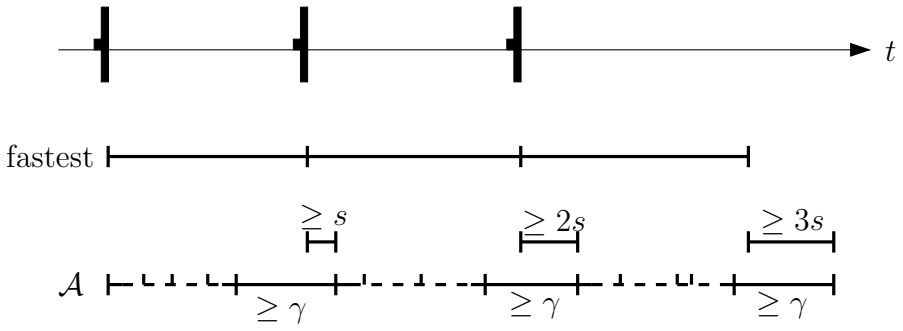

Figure 9: A sequence of bursts.

holds, we know that $\mathcal{A}$ cannot use solution Π_{FAST}: if $\mathcal{A}$ used Π_{FAST} and no more jobs followed after the burst, then $\mathcal{A}$'s competitive ratio would be larger than k. And since

$$\frac{\mathcal{F}_{\mathcal{A}}^{\text{FAST}}}{\mathcal{F}_{\mathcal{A}}^{\text{SLOW}}} = \frac{\mathcal{F}_{\mathcal{A}}^{prev} + r^3 + 2r^2 + sr + r + s}{\mathcal{F}_{\mathcal{A}}^{prev} + 2r^2 + r + sr + 2s}$$

grows linearly with r ($\mathcal{F}_{\mathcal{A}}^{prev}$ is constant w.r.t. r), there is always a value for r that satisfies the inequality. Hence, using such a value for r yields a burst which $\mathcal{A}$ must split into at least two batches. $\mathcal{A}$ thus finishes processing the last job of the burst at least s later than the fastest possible solution.

In order to force $\mathcal{A}$ into a delay of γ compared to the fastest possible solution, we construct an instance containing *a sequence of bursts*. However, care must be taken in doing so; the value of r must be chosen suitably for each burst, and the arrivals of the bursts have to be placed wisely. These details are explained in the following.

For each burst, two conditions must hold:

1. $\mathcal{A}$ cannot bundle the whole burst into one batch. Let r' be the smallest value for which
$$\frac{\mathcal{F}_{\mathcal{A}}^{\text{FAST}}}{\mathcal{F}_{\mathcal{A}}^{\text{SLOW}}} > k$$
is satisfied.

2. The large job at the end of the burst takes processing time at least γ. This requires that $r^2 + s \geq \gamma$, which is equivalent to $r \geq \sqrt{\gamma - s}$.

For each burst, we choose $r = \max\{r', \sqrt{\gamma - s}\}$, such that both of the described conditions hold.

The first burst is placed at time 0. Each subsequent burst is placed exactly at the point in time where the fastest possible solution finishes processing the previous burst (see Figure 9). We keep adding bursts to the instance until either

- $\mathcal{A}$ is "behind" so much that at the point in time where the fastest solution has finished processing the current burst, $\mathcal{A}$ has not even started processing the last job of the burst, or

- the number of bursts h is large enough such that $h \cdot s \geq \gamma$.

In both cases, $\mathcal{A}$ has a delay of γ compared to the fastest possible solution, as we will see below.

Once that one of the two situations just described is reached, a large number of small jobs is placed immediately after (the time when) $\mathcal{A}$ starts processing the last job of the last

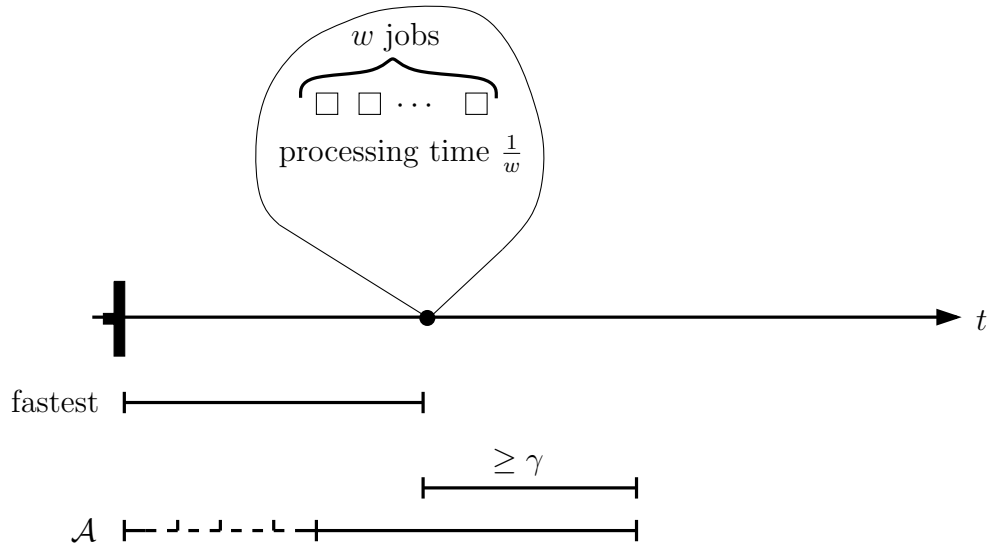

Figure 10: Many small jobs after a burst sequence.

burst[2]. More precisely, w jobs with processing time $\frac{1}{w}$ are used, as shown in Figure 10. The value of w will be chosen later. All w small jobs inevitably need to wait until processing the last job of the bursts has finished, which takes time $r_{last}^2 + s \geq \gamma - s + s = \gamma$. After that, the costs for processing the w jobs optimally are at least

$$\frac{1}{2}\frac{1}{w}w^2 + sw = \frac{1}{2}w + sw$$

according to Observation 5. Hence, the costs for processing the w last jobs are at least

$$\overbrace{\gamma w}^{\text{waiting}} + \frac{1}{2}w + sw,$$

so the total costs of $\mathcal{A}$ for this instance are

$$\mathcal{F}_{\mathcal{A}} = \mathcal{F}_{\mathcal{A}}^{bursts} + \gamma w + \frac{1}{2}w + sw.$$

In comparison, the costs of the optimal solution can be bounded by the solution that processes all bursts in the fastest possible way, and finally bundles the w small jobs together into one batch. Therefore,

$$\mathcal{F}_{\text{OPT}} \leq \mathcal{F}_{\text{FAST}}^{bursts} + w(w\frac{1}{w} + s) = \mathcal{F}_{\text{FAST}}^{bursts} + w + sw$$

so the competitive ratio of $\mathcal{A}$ is bounded by

$$\frac{\mathcal{F}_{\mathcal{A}}}{\mathcal{F}_{\text{OPT}}} \geq \frac{\mathcal{F}_{\mathcal{A}}^{bursts} + \gamma w + \frac{1}{2}w + sw}{\mathcal{F}_{\text{FAST}}^{bursts} + w + sw}.$$

By choosing w large enough, this ratio becomes arbitrarily close to

$$\frac{\gamma + \frac{1}{2} + s}{1 + s} > \frac{\gamma}{1 + s}.$$

[2]In both situations, the fastest solution has already finished processing all previous bursts by that time.

Precisely, for the ratio to be

$$\geq \frac{\gamma}{1+s} - \epsilon,$$

set

$$w \geq 2\frac{\mathcal{F}^{bursts}_{\text{FAST}}(\gamma - \epsilon(1+s)) - \mathcal{F}^{bursts}_{\mathcal{A}}(1+s)}{1+3s+2s^2+2\epsilon+4\epsilon s+2\epsilon s^2}.$$

Finally, if we set $\epsilon = 1$ and $\gamma = (k+1)(1+s)$, we obtain an instance for which the competitive ratio of $\mathcal{A}$ is larger than k. □

3 The offline BatchFlow problem: order makes a difference

We now move to the offline setting. First, we consider the offline BatchFlow problem under the constraint that jobs can only be processed in release time order, and present an $O(n^5)$ algorithm for this case. Then, Section 3.2 shows that the general offline BatchFlow problem is NP-complete, even with batch setup time equal to zero, and no constraint on the maximum flow time.

3.1 The offline BatchFlow problem for a fixed job sequence is in P

Under the fixed job sequence restriction, where jobs are to be processed in release order, the offline BatchFlow problem is solvable in polynomial time by a dynamic programming approach.

The dynamic program considers a partial solution for the first i jobs $1, 2, \ldots, i$, ignoring all subsequent jobs. Such a partial solution has two important characteristics: (i) the total flow time of the partial solution, that is, the sum of the flow times $\sum_{j=1}^{i} F_j$ of the i jobs, and (ii) the point in time when all i jobs have been processed and the machine becomes idle again, which we call the *makespan* of the partial solution. The makespan of a partial solution is important because it affects the choices for later decisions, and through that also the waiting times of the jobs after job i. Our approach is based on the observation that any solution that processes jobs $1, 2, \ldots, i$ with makespan t or earlier can be combined with any solution for jobs $i+1, i+2, \ldots, n$ that starts processing batches at time t or later. Our dynamic programming approach uses a forward recursion, where batches are appended to partial solutions.

The size of the last batch in a partial solution

Consider an optimal partial solution for jobs $1, \ldots, i$ with makespan t, and with the last batch σ_u containing exactly j jobs. Such an optimal solution can be constructed as follows: Find the optimal solution for the jobs $1 \ldots, i-j$ that finishes at $t' \leq t - P_u$, and add a last batch $\sigma_u = \{i-j+1, \ldots, i\}$, as shown in Figure 11. The last batch is started at time $t - P_u$, such that the obtained solution finishes exactly at time t. Note that this is only possible if the last job i of batch σ_u has arrived by $t - P_u$. Let $\mathcal{F}[i,t]$ be the total flow time of an optimal solution for jobs $1, 2, \ldots, i$ with makespan t, and set $\mathcal{F}[i,t] = \infty$ if no such solution exists. By trying each possible j for the size of the last batch $\sigma_u = \{i-j+1, \ldots, i\}$, and different makespans for solutions for jobs $1, \ldots, i-j$, we have

$$\mathcal{F}[i,t] = \min_{j=1..i} \Big\{ \min_{t' \leq t-P_u} \big\{ \mathcal{F}[i-j, t'] + \sum_{l=i-j+1}^{i} (t - r_l) + j \cdot P_u \big\} \Big\} \quad \text{if } r_i \leq t - P_u, \qquad (4)$$

where for *each* job $l \in \sigma_u$, the flow time of job l equals $t - r_l$. The recursion (4) forms the basis of our dynamic program.

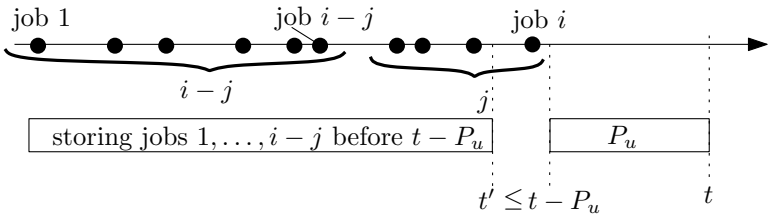

Figure 11: Extending a partial solution.

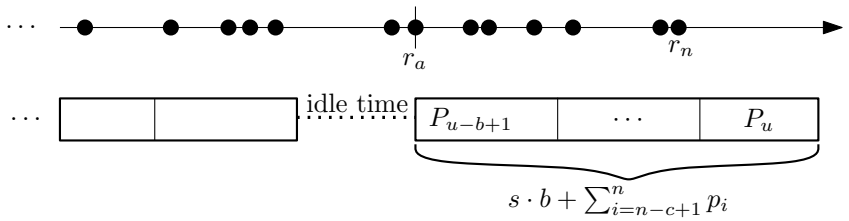

Figure 12: The makespan of a solution Π, with batches represented by boxes.

The makespan of an optimal solution

For an efficient algorithm, we must limit the number of values of t for which $\mathcal{F}[i,t]$ is evaluated. We now explain how to achieve this.

Lemma 13. *For the offline* BatchFlow *problem with a fixed job sequence, the makespan of any optimal solution can be written as*

$$r_a + b \cdot s + \sum_{i=n-c+1}^{n} p_i,$$

for some $a \in \{1,\ldots,n\}$, $c \in \{1,\ldots,n\}$ *and* $b \in \{1,\ldots,c\}$.

Proof. Consider any optimal solution Π. Further consider the last time interval that the machine is idle before a batch in Π starts (see Figure 12). Such an interval exists because there is idle time by definition before the first job arrives. Since Π is optimal, the first batch after that idle time will start exactly at the release time r_a of some job a. Let c be the number of jobs that have not been processed by time r_a. Starting at time r_a, these c jobs $n-c+1,\ldots,n$ are processed in a continuous sequence of b batches $\sigma_{k-b+1},\ldots,\sigma_k$ without idle times in between. The total processing time of these b batches is $\sum_{i=1}^{b} P_{k-b+i} = bs + \sum_{i=n-c+1}^{n} p_i$, which concludes the proof. □

We call any (partial) solution with a makespan of the form described in Lemma 13 a *candidate solution*. With Lemma 13, the following observation is immediate.

Observation 14. *The number of different makespans for all candidate solutions is at most* n^3.

An $O(n^5)$ algorithm

Lemma 13 shows that we can replace t in the definition of $\mathcal{F}[i,t]$ with the three parameters a,b,c which together describe any relevant makespan t. In other words, define $\mathcal{F}[i,a,b,c]$ as the minimal total job flow time of a solution whose last idle time ends at r_a, after which the last c jobs are processed in a continuous sequence of b batches (or ∞ if no such solution exists). Since all $i,a,b,c \in \{1,2,\ldots,n\}$, there are at most n^4 values of $\mathcal{F}[i,a,b,c]$.

These values can be computed by dynamic programming, formulating the problem as a directed shortest path problem where each $\mathcal{F}[i, a, b, c]$ is represented by a node, and edges represent feasible extensions of partial solutions. Instead of describing this dynamic program in detail, the algorithm EvalRecursion presents an alternative solution that directly follows the recursive formulation to find an optimal solution, but considers only candidate solutions.

Theorem 15. EvalRecursion *finds an optimal solution to the offline* BatchFlow *problem with a fixed job sequence in time* $O(n^5)$.

Proof. We show that for each $i = 1, \ldots, n$, the set $solutions(i)$ contains only candidate solutions, and specifically, for each makespan of a candidate solution, a solution with minimal flow time: For a single job, there is only one possible solution, namely processing job 1 separately, which the algorithm finds. By induction, suppose that for $l = 1, \ldots, i-1$, the set $solutions(l)$ contains, for each makespan of a candidate solution, a solution with minimal flow time. Let j^* be the size of the last batch in an optimal partial solution Π for jobs $1, \ldots, i$. As the algorithm tries all possible values for j, it also tried j^*. Therefore, it found that optimal solution Π, and either added it to $solutions(i)$, or there was already another solution in $solutions(i)$ with equal makespan and flow time when Π was considered. Furthermore, $solutions(i)$ contains only candidate solutions, because appended batches always start as early as possible.

The algorithm's running time can be bounded by $n \cdot n \cdot (\max_{i=1,\ldots,n} |solutions(i)|) = O(n \cdot n \cdot n^3) = O(n^5)$, using Observation 14 to bound the number of solutions that can be in any set $solutions(i)$ by n^3. □

Algorithm EvalRecursion: Solves the offline BatchFlow problem with a fixed job sequence, in time $O(n^5)$.

```
solutions(1) = {(batch job 1 separately)}
for i = 2 to n do
    solutions(i) = {}
    for j = 1 to i do
        for each solution x in solutions(i − j) do
            x_new = solution created by appending to x a batch of size j, starting at
            the earliest possible time
            t_new = makespan of x_new
            if no solution in solutions(i) with makespan t_new then
                add x to solutions(i)
            else
                x_old = solution in solutions(i) with makespan t_new
                if flow time of x_new < flow time of x_old then
                    replace x_old in solutions(i) by x_new
                end
            end
        end
    end
end
return a solution in solutions(n) with smallest flow time
```

Variations for a fixed job sequence

For simplicity, we have so far ignored the maximum flow time constraint F_{max}. However,

both the dynamic program and Algorithm EvalRecursion can be modified easily to find optimal feasible solutions under that constraint: whenever searching for the best of a set of (possibly partial) solutions, ignore any solution which violates the F_{max} constraint. In addition, it is no problem to allow only limited batch sizes.

An $O(n^2)$ dynamic program for the offline BatchFlow problem for a fixed job sequence and identical release times

In this section, we consider the offline BatchFlow problem identical release times and a fixed job sequence. For this special case, there is a simple dynamic program which computes the optimal offline solution in $O(n^2)$ time.

In Section 3.1, both costs and makespan had to be considered when comparing two partial solutions. When all jobs arrive at the same time, the makespan can be converted into an additional cost, such that a single cost value can be used for comparing two partial solutions.

Theorem 16. *Let $(\mathcal{F}_1, t_1)$ and $(\mathcal{F}_2, t_2)$ be costs and makespan of two partial solutions Π_1 and Π_2 for jobs $1, \ldots, i$. If $\mathcal{F}_1 + (n-i)t_1 < \mathcal{F}_2 + (n-i)t_2$, then Π_2 cannot be part of any optimal solution.*

Proof. In contradiction, assume there is an optimal solution Π_{OPT} where jobs $m_1, \ldots, m_i$ are batched as in Π_2. If we change the solution Π_{OPT} such that $m_1, \ldots, m_i$ are batched as in Π_1, the costs of this new solution Π_* are

$$\mathcal{F}(\Pi_*) = \mathcal{F}(\Pi_{\text{OPT}}) + (n-i) \cdot (t_1 - t_2) + (\mathcal{F}_1 - \mathcal{F}_2),$$

because all jobs $m_{i+1}, \ldots, m_n$ need to wait $(t_1 - t_2)$ longer in the new solution, and the costs for processing jobs $m_1, \ldots, m_i$ is increased by $(\mathcal{F}_1 - \mathcal{F}_2)$. Thus,

$$\mathcal{F}(\Pi_*) < \mathcal{F}(\Pi_{\text{OPT}}) \Leftrightarrow (n-i) \cdot (t_1 - t_2) + (\mathcal{F}_1 - \mathcal{F}_2) < 0$$

$$\Leftrightarrow (n-i) \cdot t_1 + \mathcal{F}_1 < \mathcal{F}_2 + (n-i) \cdot t_2.$$

So if $\mathcal{F}_1 + (n-i)t_1 < \mathcal{F}_2 + (n-i)t_2$, then Π_* has lower costs than Π_{OPT}, which contradicts our assumption. □

Theorem 16 shows that partial solutions for the same jobs can be compared by a single value. Algorithm 2 uses this observation to compute the optimal solution in $O(n^2)$ time. The variable $\mathcal{F}[i]$ in the algorithm contains for each i the $cost + (n-i) \cdot makespan$ value of the best solution for messages $m_1, \ldots, m_i$. In line 6, this value is used to incrementally compute the value of a new solution. After executing Algorithm 2, the array $index[]$ contains all information on the optimal solution. Algorithm 3 does the backtracking typical to dynamic programs, which prints the batch sizes of the optimal solution in the correct order.

Note that the same principle works for arbitrary processing times, provided that the processing order of the jobs is fixed. To that end, the line $cost = \mathcal{F}[i-j] + j(jp+s) + (jp+s)(n-i)$ is replaced by

$$cost = \mathcal{F}[i-j] + j(p_{i-j+1}, p_{i-j+2}, \ldots, p_i + s) + (p_{i-j+1}, p_{i-j+2}, \ldots, p_i + s)(n-i).$$

However, since computing *cost* in this way requires up to n computation steps, this would increase the running time to $O(n^3)$. Therefore, we compute the batch processing time incrementally, as described in Algorithm 4, to keep the $O(n^2)$ running time.

Algorithm 2: An $O(n^2)$ algorithm for the offline BatchFlow problem, restricted to identical release times and identical processing times.

```
F[0] = 0
for i = 1 to n do
    best = ∞
    index[i] = undefined
    for j = 1 to i do
        cost = F[i − j] + j(jp + s) + (jp + s)(n − i)
        if cost < best then
            best = cost
            index[i] = j
        end
    end
    F[i] = best
end
```

Algorithm 3: Extracting the optimal solution from array *index*[] by backtracking.

```
println("The optimal batch sizes in correct order are:")
i = n
k = 1
while i ≠ index[i] do
    size[k] = index[i]
    k = k + 1
    i = i − index[i]
end
for i = 1 to k do
    println(size[i])
end
```

3.2 The offline BatchFlow problem is NP-complete

If there is no setup time ($s = 0$), there is an optimal solution which processes all jobs separately. Therefore, finding an optimal solution boils down to finding an optimal permutation of the jobs $1, \ldots, n$. This problem is identical to non-preemptive scheduling on a single machine with release times, where the average flow time is minimized, denoted by $1|r_j|\sum F_j$. The latter scheduling problem is NP-complete (Lenstra et al., 1977). Hence, it follows that the general offline BatchFlow problem is NP-complete.

Theorem 17. *The offline* BatchFlow *problem is NP-complete, even with setup time* $s = 0$ *and* $F_{max} = \infty$.

In the above problem reduction, the costs of a solution for $1|r_j|\sum F_j$ equal the total job flow time $\mathcal{F}$ of the corresponding offline BatchFlow problem solution. This allows adopting the non-approximability result for $1|r_j|\sum F_j$ given in Kellerer et al. (1999) to the offline BatchFlow problem with $s = 0$, which gives the following corollary.

Corollary 18. *No polynomial-time approximation algorithm for the general offline* BatchFlow *problem can have a worst-case performance guarantee of* $O(n^{1/2-\epsilon})$ *for any* $\epsilon > 0$, *unless P=NP.*

Algorithm 4: An $O(n^2)$ algorithm for the offline BatchFlow problem restricted to identical release times and fixed job sequence.

```
F[0] = 0
for i = 1 to n do
    best = ∞
    index[i] =undefined
    batch_proc_time = s
    for j = 1 to i do
        batch_proc_time = batch_proc_time + p_{i−j+1}
        cost = F[i − j] + j · batch_proc_time + (n − i) · batch_proc_time
        if cost < best then
            best = cost
            index[i] = j
        end
    end
    F[i] = best
end
```

4 Conclusion and open problems

We considered the BatchFlow problem, which is written as the family scheduling with batching problem $1|r_j, s_f = s, F = 1|\sum F_j$ in the standard scheduling classification scheme. For this problem, we considered both offline and online variants, with restrictions on the processing times, the reordering of jobs, and the maximum flow time of a single job.

For the online BatchFlow problem with identical processing times, we presented a 2-competitive greedy algorithm, as well as two lower bounds for any online algorithm. We also derived bounds for the general online BatchFlow problem. As summarized in Table 1, upper and lower bound are of the same order in the case of the general online BatchFlow problem (with arbitrary p_j). In the case of identical processing times ($p_j = p$ for all j), bounds are only matching if $s \gg p$, as then $1+\frac{1}{1+2\frac{p}{s}}$ is close to 2. Further, we showed that no online algorithm can guarantee to compute a feasible solution for the online BatchFlow problem with a maximum processing time constraint whenever there is a feasible offline solution. We showed that the general offline BatchFlow problem is NP-complete, even

online BatchFlow problem	lower bound	upper bound
general p_j	$\frac{n}{2} - \epsilon$	n
identical $p_j = p$	$1 + \frac{1}{1+2\frac{p}{s}}$, $1 + \frac{1}{1+\frac{\max(p,s)}{\min(p,s)}}$	2

Table 1: Results for the online BatchFlow problem.

with machine setup time $s = 0$ and without a maximum flow time constraint. For the fixed job sequence case, we presented an $O(n^5)$ dynamic programming-like algorithm. It is open whether the running time $O(n^5)$ of our Algorithm EvalRecursion is optimal for the offline BatchFlow problem with fixed job sequence. For the NP-complete variants of the offline BatchFlow problem, the search for approximation algorithms might be interesting, especially when comparing such results with their scheduling equivalents.

In the online setting, even though our greedy algorithm is nearly worst-case optimal for identical processing times, and not too far from optimal optimal for the general online BatchFlow problem, one might expect that online algorithms which wait at least some short time may be superior. An average case analysis for the online BatchFlow problem,

assuming random release times, might lead to further insights in this question. Along with an average case analysis, one could consider randomized online algorithms. In conclusion, even though a number of properties of the BatchFlow problem could be proved in this paper, many aspects remain to be further investigated.

References

S. Albers and P. Brucker. The complexity of one-machine batching problems. *Discrete Applied Mathematics*, 47:87–107, 1993.

P. Brucker, A. Gladky, H. Hoogeveen, M. Kovalyov, C. Potts, T. Tautenhahn, and S. van de Velde. Scheduling a batching machine. *Journal of Scheduling*, 1:31–54, 1998.

P. Brucker and M. Kovalyov. Single machine batch scheduling. *Mathematical Methods of Operations Research*, 43:1–8, 1996.

B. Chen, X. Deng, and W. Zang. On-line scheduling a batch processing system to minimize total weighted job completion time. *Journal of Combinatorial Optimization*, 8:85–95, 2004.

T. Cheng, V.S. Gordon, and M. Kovalyov. Single machine scheduling with batch deliveries. *European Journal of Operational Research*, 94:277–283, 1996.

T. Cheng and M. Kovalyov. Single machine batch scheduling with sequential job processing. *IIE Transactions on Scheduling and Logistics*, 33:413–420, 2001.

E. Coffman, M. Yannakakis, M. Magazine, and C. Santos. Batch sizing and job sequencing on a single machine. *Annals of Operations Research*, 26:135–147, 1990.

X. Deng, H. Feng, P. Zhang, Y. Zhang, and H. Zhu. Minimizing mean completion time in a batch processing system. *Algorithmica*, 38(4):513–528, 2004.

X. Deng, C.K. Poon, and Y. Zhang. Approximation algorithms in batch processing. *Journal of Combinatorial Optimization*, 7:247–257, 2003.

S. Divakaran and M. Saks. Online scheduling with release times and set-ups. Technical Report 2001-50, DIMACS, 2001.

L. Epstein and R. van Stee. Lower bounds for on-line single-machine scheduling. *Theoretical Computer Science*, 299(1-3):439–450, 2003.

A. Janiak, M. Kovalyov, and M. Portmann. Single machine group scheduling with resource dependent setup and processing times. *European Journal of Operational Research*, 162: 112–121, 2005.

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

M. Kovalyov, C. Potts, and V. Strusevich. Batching decisions for assembly production systems. *European Journal of Operational Research*, 157:620–642, 2004.

C.Y. Lee, R. Uzsoy, and L.A. Martin-Vega. Efficient algorithms for scheduling semiconductor burn-in operations. *Operations Research*, 40(4):764–775, 1992.

J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. In P.L. Hammer, E.L. Johnson, B.H. Korte, and G.L. Nemhauser, editors, *Studies in Integer Programming*, volume 1, pages 343–362. North-Holland, 1977.

C.K. Poon and W. Yu. On minimizing total completion time in batch machine scheduling. *International Journal of Foundations of Computer Science*, 15(4):593–607, 2004.

C.K. Poon and W. Yu. A flexible on-line scheduling algorithm for batch machine with infinite capacity. *Annals of Operations Research*, 133:175–181, 2005a.

C.K. Poon and W. Yu. On-line scheduling algorithms for a batch machine with finite capacity. *Journal of Combinatorial Optimization*, 9:167–186, 2005b.

C. Potts and M. Kovalyov. Scheduling with batching: A review. *European Journal of Operational Research*, 120:228–249, 2000.

S. Webster and K.R. Baker. Scheduling groups of jobs on a single machine. *Operations Research*, 43(4):692–703, 1995.

G. Zhang, X. Cai, and C.K. Wong. On-line algorithms for minimizing makespan on batch processing machines. *Naval Research Logistics*, 48:241–258, 2001.