A Dynamic Programming Framework for Non-Preemptive Scheduling Problems on Multiple Machines

[Extended Abstract]

Sungjin Im[*] Shi Li [†] Benjamin Moseley [‡] Eric Torng[§]

Abstract

In this paper, we consider a variety of scheduling problems where n jobs with release times are to be scheduled *non-preemptively* on a set of m identical machines. The problems considered are machine minimization, (weighted) throughput maximization and min-sum objectives such as (weighted) flow time and (weighted) tardiness.

We develop a novel quasi-polynomial time dynamic programming framework that gives $O(1)$-speed $O(1)$-approximation algorithms for the offline versions of machine minimization and min-sum problems. For the weighted throughput problem, the framework gives a $(1+\epsilon)$-speed $(1-\epsilon)$-approximation algorithm. The generic DP is based on improving a naïve exponential time DP by developing a sketching scheme that compactly and accurately approximates parameters used in the DP states. We show that the loss of information due to the sketching scheme can be offset with limited resource augmentation.This framework is powerful and flexible, allowing us to apply it to this wide range of scheduling objectives and settings. We also provide new insight into the relative power of speed augmentation versus machine augmentation for non-preemptive scheduling problems; specifically, we give new evidence for the power and importance of extra speed for some non-preemptive scheduling problems.

This novel DP framework leads to many new algorithms with improved results that solve many open problems, albeit with quasi-polynomial running times. We highlight our results as follows. For the problems with min-sum objectives, we give the first $O(1)$-speed $O(1)$-approximation algorithms for the multiple-machine setting. Even for the single machine case, we reduce both the resource augmentation required and the approximation ratios. In particular, our approximation ratios are either 1 or $1+\epsilon$. Most of our algorithms use speed $1+\epsilon$ or $2+\epsilon$. We also resolve an open question (albeit with a quasi-polynomial time algorithm) of whether less than 2-speed could be used to achieve an $O(1)$-approximation for flow time. New techniques are needed to address this open question since it was proven that previous techniques are insufficient. We answer this open question by giving an algorithm that achieves a $(1+\epsilon)$-speed 1-approximation for flow time and $(1+\epsilon)$-speed $(1+\epsilon)$-approximation for weighted flow time.

For the machine minimization problem, we give the first result using constant resource augmentation by showing a $(1+\epsilon)$-speed 2-approximation, and the first result only using speed augmentation and no additional machines by showing a $(2+\epsilon)$-speed 1-approximation. We complement our positive results for machine minimization by considering the discrete variant of the problem and show that no algorithm can use speed augmentation less than $2^{\log^{1-\epsilon} n}$ and achieve approximation less than $O(\log\log n)$ for any constant $\epsilon > 0$ unless NP admits quasi-polynomial time optimal algorithms. Thus, our results show a stark contrast between the two settings. In one, constant speed augmentation is sufficient whereas in the other, speed augmentation is essentially not effective.

[*]Electrical Engineering and Computer Science, University of California, 5200 N. Lake Road, Merced CA 95344. sim3@ucmerced.edu. Partially supported by NSF grants CCF-1008065 and 1409130. Part of this work was done when the author was at Duke University.

[†]Toyota Technological Institute at Chicago, 6045 S. Kenwood Ave. Chicago, IL 606371. shili@ttic.edu.

[‡]Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis MO, 63130, USA. bmoseley@wustl.edu.

[§]Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824. torng@msu.edu.

1 Introduction

In this paper, we present a new dynamic programming framework that provides new effective algorithms for a wide variety of important non-preemptive scheduling problems. For a typical problem that we study, the input instance consists of a set of n jobs that arrive over time. In all but the machine minimization problem, we are also given the number m of identical machines on which we can schedule jobs. Each job J has a release (or arrival) time r_J, a processing time p_J and, depending on the exact problem definition, may have a deadline d_J or a weight w_J. In the unweighted version of a problem, all jobs have weight 1. When a job J is scheduled, it must be scheduled for p_J consecutive time steps after r_J on a machine. Let C_J be the completion time of job J under some schedule. The flow time of job J is defined to be $F_J = C_J - r_J$. Using our dynamic programming framework, we develop new algorithms and results for the following collection of problems. If the short name of a problem starts with W, then jobs have weights.

- **Machine Minimization** (MM): Jobs have deadlines and no weights. The goal is to schedule all jobs by their deadline using the minimum number of machines.
- **(Weighted) Throughput Maximization** (WThr, Thr): Jobs have deadlines and not all jobs need to be scheduled. The goal is to maximize the total weight of the jobs scheduled by their deadline.
- **Total (Weighted) Flow Time** (WFT, FT): Jobs have no deadline. The objective is $\min \sum_J w_J F_J$.
- **Total (Weighted) Tardiness** (WTar, Tar): Jobs have deadlines but they do not need to be completed by their deadlines. The objective is $\min \sum_J w_J \max\{(C_J - d_J), 0\}$.

All of these problems are NP-hard even on a single machine [15]; NP-hardness holds for the preemptive versions of these problems when we consider multiple machines. More prior work has been done on the preemptive versions of these problems (see [23, 19] for pointers to some of this work) than the non-preemptive versions. One possible reason for this is the challenge of identifying effective bounds on the value of the optimal non-preemptive solution for these problems. Finding effective bounds on the optimal preemptive solution for these problems, while also difficult, is easier. One of the key contributions of our dynamic programming framework is that we are able to provide effective bounds that allow the development of approximation algorithms with small approximation ratios.

Here is a brief summary of prior work on these non-preemptive problems. For FT, WFT, Tar and WTar (we refer to these problems as the min-sum problems), there are very strong lower bounds. Specifically, it is NP-hard to get $o(\sqrt{n})$-approximations for these problems [22] (Tar (WTar) is harder than FT (WFT) since by setting $d_J = r_J$, the Tar (WTar) problem becomes the FT (WFT) problem). For MM, randomized rounding [24] leads to an $O(\log n/\log\log n)$-approximation. The approximation ratio gets better as opt gets larger. This was the best known algorithm until a breakthrough of Chuzhoy et al. [11] that showed an $O(\mathsf{opt})$-approximation, where opt is the optimum number of machines needed. That is, the algorithm uses $O(\mathsf{opt}^2)$ machines. This implies an $O(1)$-approximation when $\mathsf{opt} = O(1)$. Combining this and the randomized rounding algorithm gives an $O(\sqrt{\log n/\log\log n})$-approximation. This is currently the best known result for this problem (the $O(1)$-approximation result [9] is unfortunately incorrect [10]). For Thr and WThr, several $\Omega(1)$ approximations are known [5, 13]. (We use the convention that approximation ratios for maximization problems are at most 1.) In particular, the best approximation ratio for both problems is $1 - 1/e - \epsilon$.

Given the strong lower bounds, particularly for the min-sum objective problems and MM, we are forced to relax the problem to derive practically meaningful results. One popular method for doing this is to use resource augmentation analysis where the algorithm is given more resources than the optimal solution it is compared against [21]; specifically machine augmentation (extra machines), speed augmentation (faster machines), or both machine and speed augmentation (extra and faster machines). Bansal et al. [3] applied resource augmentation to most of the above problems with $m = 1$ (or $\mathsf{opt} = 1$ in MM, where opt is the optimum number of machines needed). Table 1 shows their results. For FT, WFT and Tar, they gave 12-speed 2-approximation algorithms. For MM and Thr, they gave 24-speed 1-approximations. Their work introduced an interesting linear program for these problems and rounded the linear program using speed augmentation. Their work, unfortunately, does not seem to generalize to the multiple machine setting even if there are $O(1)$ machines. We also note that their techniques cannot be leveraged to obtain $O(1)$-approximations for the min-sum objectives with less than 2-speed because their linear program has a large integrality gap with less than 2-speed.

We are motivated by the following open problems, some more general in nature and others problem specific. On the general side, we have two main questions. First, how can we develop effective lower bounds on the optimal solution for a given non-preemptive scheduling instance? Second, what is the relative power of speed augmentation versus machine augmentation for non-preemptive scheduling problems. On the problem specific side, we strive to answer the following open questions. Can one use $O(1)$-speed to get an $O(1)$-approximation (or even 1-approximation) for MM when $\mathsf{opt} > 1$? For the min-sum problems, what can be shown when $m > 1$? Finally, can we achieve an $O(1)$-approximation for any min-sum problem while using less than 2-speed? This is an open question even when $m = 1$.

1.1 Our Contributions In this paper, we present new results for a wide variety of non-preemptive job scheduling problems. Our results follow from a novel general dynamic programming framework for scheduling problems; we discuss the high-level ideas and novelty of our framework in Section 1.2. Our framework is flexible enough to give improved results, summarized in Table 1, for the above-mentioned scheduling problems. The main drawback of our algorithms is that they run in quasi-polynomial time. However, our algorithms have many merits that we highlight below.

- The speed factors and approximation ratios are much smaller than those of prior work. We get 1-approximation ratio for MM, Thr and FT and $(1+\epsilon)$-approximation for WThr, WFT and WTar. For MM, Thr, WThr, FT and WFT, our speed factor is either $1 + \epsilon$ or $2 + \epsilon$.
- Our algorithms work when m, the number of machines, is big (or opt is big in the MM problem).
- Our DP framework is very flexible. In Section 8, we show how it can handle a variety of scheduling problems, in addition to the main problems we are considering.
- We provide new evidence for the power of extra speed for non-preemptive scheduling. In the preemptive setting, speed dominates machine augmentation, but intuitively machine augmentation could be more useful in non-preemptive settings. Our DP framework delivers the following information. By using $(1+\epsilon)$-speed, the scheduling problems become much simpler. In any of our results where we require $(2 + \epsilon)$-speed, we can replace the speed augmentation with a $(1+\epsilon)$-speed 2-machine algorithm[1]. Note that we always require $(1 + \epsilon)$-speed. Thus, other than the $(1 + \epsilon)$-speed, our results show that speed-augmentation and machine-augmentation have similar power.

Besides these general properties, we resolve the following open problems, albeit with quasi-polynomial time algorithms.

- For FT, it was open if one could get an $O(1)$-approximation with speed less than 2, as previous techniques were shown not to be useful without 2-speed [3]. This was open even when $m = 1$. Our new techniques yield a $(1 + \epsilon)$-speed 1-approximation for FT for general m, solving this open question. For WFT, we get $(1 + \epsilon)$-speed $(1 + \epsilon)$-approximation.
- For MM when opt > 1, we give the first $O(1)$-approximation using $O(1)$-speed.

We complement our results for MM by considering the more general *discrete* variant of the problem. The previously mentioned version of the problem is called the *continuous* variant. In the discrete variant, a job J has a set of intervals I_J where the job can be feasibly scheduled and these intervals need not be contiguous or overlapping. For this problem, again randomized rounding can be used to give an $O(\log n/ \log\log n)$-approximation. It is also known that there is no $O(\log\log n)$-approximation for the problem unless $\mathtt{NP} \subseteq n^{O(\log\log\log n)}$ [12]. We extend the work of [12] to show that speed is essentially not helpful in this case. Specifically, we show there is no polynomial time $o(\log\log n)$-approximation for the problem using $O(2^{\log^{1-\epsilon} n})$-speed for any constant $\epsilon > 0$ unless $\mathtt{NP} \subseteq n^{\mathtt{poly}(\log n)}$. This result is briefly discussed in Section 7; the complete proof will appear in the full version of this paper. It shows a stark contrast between the two versions of the problem as in the continuous version, speed-augmentation is very useful.

1.2 Technical Contributions and Novelty of our Dynamic Programming One of the key roadblocks for any improved result in MM is developing a provable lower bound for the given instance. Chuzhoy et al. observed that the standard linear programming formulation has an $\Omega(\log n/ \log\log n)$-integrality gap [11]. Chuzhoy et al. overcame this difficulty by developing a very clever recursive LP solution where they round up the number of machines required to schedule jobs in a subproblem before solving the LP for the current problem. Unfortunately, it is not clear how to use this idea to gain a better result than their current result.

For the problems with min-sum objectives, the challenge is that previous techniques do not seem to be useful for scheduling on multiple machines. The work of [3] used a linear program similar to the one used by [11]. However, the rounding procedure used crucially relies on the linear program scheduling jobs on a single machine. In particular, on a single machine one can optimally pack jobs into a scheduling interval, but this creates non-trivial challenges on multiple machines. Further, in [3], they show their linear program has a large integrality gap with speed less than 2. Due to these and other issues with previous techniques, it was not clear how to show positive results for min-sum objectives on multiple machines and how to improve the resource augmentation required to sub-polylogarithmic in the machine minimization problem.

Our novel DP framework for the scheduling problems

[1]Our speed factors for Tar and WTar are $8 + \epsilon$; however, we believe they can be improved to $2+\epsilon$ using our framework with a more involved algorithm.

Results	Machine Minimization (MM)	Throughput (Thr)	Weighted Throughput (WThr)
[3]: $m = 1$ *	$(24, 1)$	$(24, 1)$	N/A
This paper: $m \geq 1$	$(1+\epsilon, 2)$ and $(2+\epsilon, 1)$	$(1+\epsilon, 1-\epsilon)$ and $(2+\epsilon, 1)$	$(1+\epsilon, 1-\epsilon)$

	Flow Time (FT)	Weighted Flow Time (WFT)	Tardiness (Tar)	Weighted Tardiness (WTar)
[3]: $m = 1$ *	$(12, 2)$	$(12, 2)$	$(12, 2)$	$(24, 4)$ with 2 machines
This paper: $m \geq 1$	$(1+\epsilon, 1)$	$(1+\epsilon, 1+\epsilon)$	$(8+\epsilon, 1+\epsilon)$	$(8+\epsilon, 1+\epsilon)$

Table 1: Summary of our results, compared to those of [3]. In all results, the first parameter is the speed factor and the second one is the approximation ratio. Note that for MM, the approximation ratio is the number of machines. (*) The results of [3] only hold if $\mathsf{opt} = 1$ (for MM) or $m = 1$ (for other problems). We note that the results of [3] require polynomial time while ours require quasi-polynomial time.

is based on naïve exponential-time recursive algorithm. In a naïve DP, each state corresponds to a possible input to a sub-problem in the recursion. The exponential running time comes from the exponential possibilities for the inputs. We address this roadblock by developing a sketching scheme that captures the input effectively. On the one hand, the sketch of an input is short so that there are relatively few possible sketches allowing our algorithm to be efficient. On the other hand, the sketch is accurate so that we incur only a small loss in the quality of solution by using the sketch.

Let's focus on MM to illustrate another novelty of our framework. Our dynamic programming goes through $\Theta(\log n)$-levels. If we measure the loss of quality directly by the number of machines, then we will need $\Theta(\log n)$ additional machines. This is due to the rounding issue: any loss in the quality will result in using one additional machine at each level. When the optimal number of machines is small, we would only obtain an $O(\log n)$-approximation. We address this issue by using a "smooth" measure of quality. In a feasible scheduling, there is a perfect matching between the set $\mathcal{J}$ of jobs and the set $\mathcal{T}$ of intervals scheduling them (we call $\mathcal{T}$ a *signature*). During the course of our dynamic programming, we relax the perfect matching requirement: we allow an interval in $\mathcal{T}$ to be used c times, for some *real* number c. Then, we measure our loss by the parameter c, which we call *congestion parameter*. This congestion parameter will be useful in many other problems.

In sum, we develop a sketching scheme that increases the congestion by a small factor, say $1 + O(1/\log n)$. Then in our $O(\log n)$-level dynamic programming, we lose a factor of 2 in the congestion. By the integrality of matching, we obtain a feasible schedule by either doubling the speed or the number of machines. This explains our previous statement that extra speed and extra machines have similar power.

We note that using sketching schemes coupled with dynamic programs that divide the time horizon into subproblems is not a new idea in scheduling theory. Several scheduling papers such as [7, 18] use similar techniques. The main novelty of our technique is in the type of sketches we create that allow our algorithm to be useful for a variety of problems and objectives.

Organization: To illustrate our dynamic programming framework, we first showcase in Section 2 and 3 how to get $(1+\epsilon, 1-\epsilon)$-approximation for WThr. The results for Thr and MM will follow from the same framework modulo small modifications. In these sections, we only consider the case when N and W are polynomial in n. In Section 4, we show how to deal with large N and W. We give algorithms for FT and WFT in Section 5, and algorithms for Tar and WTar in Section 6. In Section 7, we give our $\Omega(\log\log n)$-hardness result for the discrete version of MM with $O(2^{\log^{1-\epsilon} n})$-speed. Finally we show how our algorithms apply to some other variants of scheduling problems, and we discuss some open problems and the limitations of our framework in Section 8.

2 Algorithms for (Weighted) Throughput and Machine Minimization: Useful Definitions and Naive Recursive Algorithm

In this section, we consider the problems MM, Thr and WThr. We define some useful concepts and give the naive algorithm which our dynamic programming is based on. The dynamic programming is given in Section 3. For convenience, we only focus on the problem WThr. With slight modifications that are described at the beginning of Section 3, we get the results for Thr and MM as well. The input to WThr consists of the set $\mathcal{J}$ of n jobs to be scheduled and m identical machines. Each job $J \in \mathcal{J}$ has release time r_J, deadline d_J, processing time/size p_J, and weight w_J. We assume all parameters are non-negative integers. The task is to schedule jobs J within their window (r_J, d_J) on one of the m machines non-preemptively, with the goal of maximizing the total weight of jobs scheduled. The time horizon we need to consider is $(0, N)$ where $N := \max_{J\in\mathcal{J}} d_J$. Let $W = \max_{J\in\mathcal{J}} w_J$ denote the maximum job weight. In this and the next section, we assume N and W are polynomial in n. The general case is handled in Section 4. Throughout the proof, let $0 < \epsilon \leq 1$ be some fixed constant.

2.1 Preprocessing

We use two standard preprocessing steps to simplify the instance.

Rounding job weights. Round each weight w_J down to the nearest integer of the form $\lceil(1+\epsilon/3)^i\rceil, i \in \mathbb{Z}$ with only a $(1+\epsilon/3)$-factor loss in the approximation ratio. The resulting $z = O(\log W/\epsilon)$ different jobs weights are indexed from 1 to z. We will refer to these z weights as weight types. If jobs are unweighted, this step is unnecessary, so we have no loss in the approximation factor.

Rounding job sizes and regularly aligning jobs. We require each job to be of some type i, specified by two integers s_i and g_i. Integer s_i is the size of type-i jobs while integer g_i defines permissible starting times for scheduling type-i jobs. More specifically, we constrain type-i jobs to start at integer multiples of g_i. We call an interval of length s_i starting at some integer multiple of g_i an *aligned* interval of type-i. We call a schedule an *aligned* schedule if all intervals used in this schedule are aligned intervals. The following lemma states the outcome of this step.

LEMMA 2.1. *With $(1+\epsilon/3)$-speed augmentation, we can assume there are at most $q = O(\log N/\epsilon)$ job types $(s_1, g_1), (s_2, g_2), \cdots, (s_q, g_q)$, and our goal is to find the optimum aligned schedule. Moreover, each job type i satisfies $s_i/g_i = O(1/\epsilon)$.*

Proof. We create the set Q of types as follows. Given $\epsilon > 0$, we first define $\epsilon' = \epsilon/3$. Let $k = O(1/\epsilon)$ be an integer such that $(1+1/k)/(1-1/k) \le 1+\epsilon' = 1+\epsilon/3$. If a job $J \in \mathcal{J}$ has size p in the original instance such that $(1+1/k)^i \le p < (1+1/k)^{i+1}$ for some integer i, we let $p' := \lceil(1+1/k)^i\rceil$. If $p' < 2k$, we add $(s, g) := (p', 1)$ to Q. Otherwise, we add $(s, g) := (p' - \lfloor p'/k\rfloor + 1, \lfloor p'/k \rfloor)$ to Q. The type of job J is defined by parameters s and g.

We first prove that there are not many types of jobs. Since the original size of a job J is an integer between 1 and N, there are at most $O(\log_{1+1/k} N) = O(k \log N) = O(\frac{1}{\epsilon}\log N)$ different values of p'. The proposition follows since each value of p' defines exactly 1 job type.

We next show that any schedule of the original instance using m machines can be converted to a new $(1+\epsilon')$-speed schedule using the same number m of machines, in which every job J is scheduled on a permissible interval. Consider each fixed job J in the given instance. Suppose J was scheduled in $(a, a+p)$ in the given schedule. We now obtain a sub-interval of $(a, a+p)$ which is a permissible interval for J. Say the original size p of J satisfies $(1+1/k)^i \le p < (1+1/k)^{i+1}$ and $p' = \lceil(1+1/k)^i\rceil \le p$. We first trim the scheduling interval from $(a, a+p)$ to $(a, a+p')$. If $p' < 2k$, then job J is already scheduled on a permissible interval, since $(a, a+p')$ is an interval of length $s = p'$ and a is a multiple of $g = 1$. If $p' \ge 2k$, then $g = \lfloor p'/k \rfloor$ and $s = p' - g + 1$. Let $a' \ge a$ be the smallest integer that is a multiple of g. Then, $a' \le a + g - 1$. Then, $(a', a+p')$ is an interval of length at least s and a' is an integer multiple of g. We trim the interval $(a', a+p')$ from the right to obtain an interval of length exactly s.

We compare p and s. Since $p < (1+1/k)^{i+1}$ and $p' \ge (1+1/k)^i$, we have $p' \ge p/(1+1/k)$. Then, $p/s \le 1+1/k$ if $p' < 2k$. If $p' \ge 2k$, $s = p' - \lfloor p'/k \rfloor + 1 \ge p' - p'/k = (1-1/k)p' \ge (1-1/k)p/(1+1/k)$. Thus $p/s \le (1+1/k)/(1-1/k)$, this is at most $1+\epsilon'$ for some sufficiently large $k = O(1/\epsilon)$. Thus, scheduling $\mathcal{J}$ using the permissible scheduling intervals only requires $(1+\epsilon')$-speed which is $(1+\epsilon/3)$-speed.

With the property that $s_i/g_i = O(1/\epsilon)$, the following corollary is immediate.

COROLLARY 2.1. *For any $i \in [q]$ and any integer time t in the time horizon $(0, N)$, the number of different aligned intervals of type-i containing t is $O(1/\epsilon)$.*

DEFINITION 2.2. *(permissible interval) For each job J of type-i, we say that an interval (a_J, b_J) is a* permissible *interval for J if (a_J, b_J) is an aligned interval of type-i and $r_J \le a_J < b_J \le d_J$.*

For weighted jobs, we overload notation and say that a type-i job J with weight type j is a type-(i, j) job where $i \in [q]$ and $j \in [z]$.

2.2 Signature Whereas our ultimate goal is an actual schedule consisting of an assignment of jobs to machines at specified times, we observe that the following *signature* is sufficient for our purposes.

DEFINITION 2.3. *(signature) A signature is a multi-set of aligned intervals.*

The following proposition emphasizes why our signature $\mathcal{T}$ is sufficient for our purposes.

PROPOSITION 2.1. *Let $\mathcal{T}$ be the multi-set of aligned intervals used in the optimum aligned schedule. Given $\mathcal{T}$, one can construct a feasible schedule as good as the optimal schedule.*

Proof. We first greedily allocate the set $\mathcal{T}$ of intervals to m machines. Hence we only need to allocate some jobs in $\mathcal{J}$ to $\mathcal{T}$. To this end, we solve a maximum-weight bipartite matching problem for the bipartite graph between $\mathcal{J}$ and $\mathcal{T}$ where there is an edge between $J \in \mathcal{J}$ and $T \in \mathcal{T}$ if T is a permissible interval for J. Each job $J \in \mathcal{J}$ has weight w_J.

The advantage of considering signatures is the following. Due to information loss in our dynamic programming solution, the signature $\mathcal{T}$ we find may not lead to an optimal schedule. However, if we allow each interval in $\mathcal{T}$ to be used c times, for some *real* number $c > 1$, we can obtain a solution that is as good as the optimum schedule. As will be discussed later, we use each interval c times with a small amount of resource augmentation; namely by using $\lceil c \rceil$ times more speed/machine augmentation, or by simply discarding $c - 1$ fraction of throughput. Thus it is convenient not to associate an interval with a specific job.

2.3 A Naïve Dynamic Programming Algorithm Our algorithm is based on improving the following naïve recursive algorithm for WThr. Initially, we are given a set $\mathcal{J}$ of jobs and a block $(0, N)$. Our goal is to maximize the total weight of jobs scheduled. We recurse by reducing the instance on $(0, N)$ to two sub-instances on $(0, C)$ and (C, N) where $C = \lfloor N/2 \rfloor$. Focus on a job $J \in \mathcal{J}$. We have three choices for job J. First, we may decide to schedule J completely in $(0, C)$ or (C, N). In this case, we pass J to the first or second sub-instance. Second, we decide to schedule J on a permissible interval (a_J, b_J) for J satisfying $a_J < C < b_J$. In this case, we pass the scheduling interval (a_J, b_J) to both sub-instances and tell the sub-instances that the interval is already reserved. Third, we may choose to discard J.

In some intermediate level of the recursion, an instance is the following. We are given a block (A, B) with $A < B$, a set $\mathcal{J}'$ of jobs that need to be scheduled in this block and a set $\mathcal{T}'$ of reserved intervals. We shall find a signature $\mathcal{T}''$ in (A, B) such that $\mathcal{T}'' \uplus \mathcal{T}'$ can be allocated in m machines and schedule some jobs in $\mathcal{J}'$ using $\mathcal{T}''$. The goal is to maximize the total weight of scheduled jobs. If $B - A \geq 2$, we let $C = \lfloor (A+B)/2 \rfloor$ and reduce the instance into two sub-instances on the two sub-blocks (A, C) and (C, B), by making choices for jobs in $\mathcal{J}'$. The two sub-instances can be solved independently and recursively. We reach the base case when $B - A = 1$. The recursion defines a binary tree of blocks where the root is $(0, N)$ and the leaves are blocks of unit length. For convenience, we call this tree the *recursion tree*.

We can naturally transform this recursive algorithm to a naïve DP where each state corresponds to a possible input to a sub-instance. This naïve DP runs in exponential time since the number of valid decisions and the number of states are both exponential. As discussed in the introduction, we shall reduce the number of states by developing a compact sketching scheme, leading to our efficient DP.

3 Algorithms for (Weighted) Throughput and Machine Minimization: Efficient Dynamic Programming

In this section we give our efficient dynamic programming for MM, Thr and WThr. To state our main lemma, we shall use the following maximum weighted matching with congestion problem.

DEFINITION 3.1. *(fractional matching and congestion) Given a set $\mathcal{J}'$ of jobs and a signature $\mathcal{T}'$, construct a graph $G = (\mathcal{J}', \mathcal{T}', E)$ where there is an edge $(J, T) \in E$ if and only if T is a permissible interval for job J. Let $c \geq 1$ be a* congestion *parameter. Consider the following maximum weighted fractional matching problem: Find a fractional matching $\{x_e\}_{e \in E}$ such that every job $J \in \mathcal{J}'$ is matched to an extent at most 1 and every interval $T \in \mathcal{T}'$ is matched to an extent at most c with the goal of maximizing the total weight of matched jobs in $\mathcal{J}'$, i.e, $\sum_{J \in \mathcal{J}'} w_J \sum_{T:(J,T) \in E} x_{(J,T)}$. Let $\mathsf{MWM}(\mathcal{J}', \mathcal{T}', c)$ denote the maximum value of the problem.*

Given a fractional matching $\{x_e\}_{e \in E}$, the extent to which job J is matched refers to the quantity $\sum_{T:(J,T) \in E} x_{(J,T)}$. The main Lemma given by the DP is the following.

LEMMA 3.2. *If there is an aligned schedule which schedules opt total weight of jobs in $\mathcal{J}$ on m machines, then we can find in $2^{\mathrm{poly}(\log n, 1/\epsilon)}$-time a signature $\mathcal{T}$ that can be allocated in m machines such that $\mathsf{MWM}(\mathcal{J}, \mathcal{T}, 1 + \epsilon/3) \geq \mathsf{opt}$.*

We use Lemma 3.2 to derive results for WThr, Thr, and MM as follows. We start with WThr. Suppose $\mathcal{T}^*$ is the signature for the optimal aligned schedule. It follows that $\mathsf{MWM}(\mathcal{J}, \mathcal{T}^*, 1) = \mathsf{opt}$. From Lemma 3.2, we can find a signature $\mathcal{T}$ such that $\mathsf{MWM}(\mathcal{J}, \mathcal{T}, 1+\epsilon/3) \geq \mathsf{opt}$. By integrality of bipartite matching, we can find a set $\mathcal{J}' \subseteq \mathcal{J}$ of jobs with total weight at least $\mathsf{opt}/(1 + \epsilon/3)$ such that $\mathcal{J}'$ can be mapped to $\mathcal{T}$ integrally with congestion 1. Combining this with the $(1 + \epsilon/3)$-speed augmentation from Lemma 2.1 and the $(1 + \epsilon/3)$-factor loss of approximation ratio from rounding job weights, we get a a $(1 + \epsilon, 1 - \epsilon)$-approximation for WThr.

For Thr and MM, we use a relaxed form of Lemma 3.2. Namely, we use congestion 2 rather than congestion $1 + \epsilon/3$ to get a signature $\mathcal{T}$ such that $\mathsf{MWM}(\mathcal{J}, \mathcal{T}, 2) \geq \mathsf{opt}$. In this case, we observe that we can achieve opt value by doubling the number of machines or speed of machines. Combining this with the $(1 + \epsilon/3)$-speed augmentation from Lemma 2.1 and remembering that we have no loss due to rounding job weights, we get a $(2 + \epsilon, 1)$-approximation for Thr. For MM, by guessing the optimal number of machines, we get $(1 + \epsilon, 2)$ and $(2 + \epsilon, 1)$-approximations.

3.1 Defining a Sub-Problem: Extended WThr Instance We now start proving Lemma 3.2. We first regulate the input of a sub-instance by restricting that a job J can only be discarded at the inclusive-minimal block

(A, B) in the recursion tree containing (r_J, d_J). Thus, if $B - A \geq 2$, then J can be discarded in (A, B) if $A \leq r_J < \lfloor (A+B)/2 \rfloor < d_J \leq B$. If $B - A = 1$ then J can be discarded if $(r_J, d_J) = (A, B)$.

With this restriction, we can characterize the properties of the input to a sub-instance. The set $\mathcal{J}'$ of jobs can be divided into two subsets. The first set contains the jobs $J \in \mathcal{J}$ such that $(r_J, d_J) \subseteq (A, B)$. Each job J in this set can only be scheduled in (A, B) and cannot be discarded at upper levels. Thus such a job J must be in $\mathcal{J}'$. The set, denoted by $\mathcal{J}_{\mathsf{in}}$, is completely determined by (A, B). The second set contains the jobs J whose window (r_J, d_J) intersect (A, B) but are not contained in (A, B). We use $\mathcal{J}_{\mathsf{up}}$ to denote this set of jobs since they are passed to (A, B) from upper levels. For each $J \in \mathcal{J}_{\mathsf{up}}$, (r_J, d_J) contains either A or B. The jobs J whose windows (r_J, d_J) are disjoint from (A, B) can not be in $\mathcal{J}'$; otherwise there is no valid solution since J can be not scheduled in (A, B) and cannot be discarded in any sub-blocks of (A, B).

We use $\mathcal{T}_{\mathsf{up}}$ to denote the set of reserved intervals from upper levels. We can ignore the intervals that do not intersect (A, B) because they do not affect the instance. For the other intervals $(a, b) \in \mathcal{T}_{\mathsf{up}}$, (a, b) cannot be completely contained in (A, B); otherwise the interval would not be decided in upper levels. To sum up, in a sub-instance, we are given

1. A block (A, B) (this determines the set $\mathcal{J}_{\mathsf{in}} := \{J \in \mathcal{J} : (r_J, d_J) \in (A, B)\}$).
2. A set $\mathcal{T}_{\mathsf{up}}$ of already allocated aligned intervals that are not contained in (A, B) and intersect (A, B).
3. A set $\mathcal{J}_{\mathsf{up}} \subseteq \mathcal{J}$ of jobs J such that (r_J, d_J) is not contained in (A, B) and intersects (A, B).

The jobs in $\mathcal{J}_{\mathsf{up}}$ must be scheduled in (A, B): if we need to discard some job J in the set, we would have done so at some upper level. We can schedule some or all jobs in $\mathcal{J}_{\mathsf{in}}$. We need to guarantee that the scheduling intervals used, union $\mathcal{T}_{\mathsf{up}}$, can be allocated on m machines. The goal of the instance is to maximize the weight of scheduled jobs in $\mathcal{J}_{\mathsf{in}}$. For convenience, we call such an instance an *extended* WThr *instance* defined by (A, B) (this defines $\mathcal{J}_{\mathsf{in}}$), $\mathcal{T}_{\mathsf{up}}$ and $\mathcal{J}_{\mathsf{up}}$. The value of the instance is the maximum weight of scheduled jobs in $\mathcal{J}_{\mathsf{in}}$. Notice that we *do not* count the weight of jobs in $\mathcal{J}_{\mathsf{up}}$.

In the case $B - A \geq 2$ and $C = \lfloor (A+B)/2 \rfloor$, we shall make a decision for each job in $\mathcal{J}_{\mathsf{up}} \cup \mathcal{J}_{\mathsf{in}}$ to reduce the instance into two sub-instances. Let $D(J)$ be the decision for J. $D(J)$ can be L (passing J to the left instance), R(passing J to the right instance), $\perp$ (discarding J) or some permissible interval (a, b) for J such that $A \leq a < C < b \leq B$. $D(J)$ is valid if

1. If $D(J) = \mathsf{L}$, then (r_J, d_J) intersects (A, C).
2. If $D(J) = \mathsf{R}$, then (r_J, d_J) intersects (C, B).
3. If $D(J) = \perp$, then $J \in \mathcal{J}_{\mathsf{in}}$ and $r_J < C < d_J$.
4. If $D(J)$ is some permissible scheduling interval (a_J, b_J) for J, then $A \leq a_J < C < b_J \leq B$.

We say the decision function D is valid if all decisions $D(J)$ are valid.

3.2 Reducing the Number of Different Inputs Using a Sketching Scheme We now describe our sketching scheme. We first show that the multi-set $\mathcal{T}_{\mathsf{up}}$ is not a concern – it has relatively few possibilities.

CLAIM 3.3. *Given (A, B), there are at most $n^{O(q/\epsilon)} = n^{O(\frac{1}{\epsilon^2} \log N)}$ possibilities for $\mathcal{T}_{\mathsf{up}}$.*

Proof. Recall that each interval $(a, b) \in \mathcal{T}_{\mathsf{up}}$ contains A or B (or both). By Corollary 2.1, there can be at most $O(1/\epsilon)$ different aligned intervals for type-i jobs that intersect A (B, resp.). Thus, there can be at most $O(q/\epsilon)$ different aligned intervals in $\mathcal{T}_{\mathsf{up}}$. Since each interval can appear at most n times in the multi-set $\mathcal{T}_{\mathsf{up}}$, the total number of multi-sets is at most $n^{O(q/\epsilon)}$, which is $n^{O(\frac{1}{\epsilon^2} \log N)}$ by Lemma 2.1.

We now deal with the set $\mathcal{J}_{\mathsf{up}}$. We cannot consider all possible sets $\mathcal{J}_{\mathsf{up}}$ since there are an exponential number of possibilities. Instead, we cluster together similar possible sets into a single set and represent this common set by an approximate description. We describe how we cluster and compress as follows. We focus on one job type and one weight at a time; let $\mathcal{J}_{i,j}$ be the jobs of type-(i, j) in $\mathcal{J}_{\mathsf{up}}$. For each job $J \in \mathcal{J}_{i,j}$, (r_J, d_J) contains A or B. If (r_J, d_J) contains A, then we say J is a left-side job; otherwise we say J is a right-side job. Since we need to schedule jobs in $\mathcal{J}_{\mathsf{up}}$ inside (A, B), we think of left-side jobs as having release time A, and right-side jobs as having deadline B. Let $\mathcal{J}^{\mathsf{L}}_{i,j}$ be the set of left-side jobs in $\mathcal{J}_{i,j}$ and $\mathcal{J}^{\mathsf{R}}_{i,j}$ be the set of right-side jobs in $\mathcal{J}_{i,j}$. To define our sketch, fix some $\delta > 0$ whose value will be decided later. Let $\Delta_0 := 0, \Delta_1 := 1$, and $\Delta_i := \lceil (1+\delta)\Delta_{i-1} \rceil$ for all integers $i \geq 2$.

DEFINITION 3.4. *(sketch) Given a set $\mathcal{J}'$ of jobs of same type and same release time (deadline, resp.) which are ordered in increasing (decreasing, resp.) order of their deadlines (release times, resp.), the left-sketch (right-sketch, resp.) of $\mathcal{J}'$, denoted as $\mathsf{sketchL}(\mathcal{J}')$ ($\mathsf{sketchR}(\mathcal{J}')$, resp.), is a vector $(t_1, t_2, \cdots, t_\ell)$ s.t.*

- *ℓ is the smallest number such that $\Delta_{\ell+1} > |\mathcal{J}'|$;*
- *for every $j \in [\ell]$, t_j is the deadline (release time, resp.) of the Δ_j-th job in the ordering.*

The loss of information in this sketch is that we only know that there are $\Delta_j - \Delta_{j-1}$ jobs in the left-sketch which have deadlines between t_{j-1} and t_j for every $j \in [\ell]$. However, there are only $(1+\delta)$ factor more jobs with deadline by t_j than those with deadline by t_{j-1}. Thus, we can schedule all the former jobs by time t_{j-1} with an increase of at most $(1+\delta)$ factor in the congestion. The effectiveness of our input sketches and solution signatures is formally stated in the following lemma.

LEMMA 3.5. *Let $\mathcal{J}_1$ and $\mathcal{J}_2$ be two disjoint sets of jobs such that jobs in $\mathcal{J}_1 \cup \mathcal{J}_2$ have the same job type i and the same release time (deadline, resp.). Moreover,* $\mathsf{sketchL}(\mathcal{J}_1) = \mathsf{sketchL}(\mathcal{J}_2)$ *(*$\mathsf{sketchR}(\mathcal{J}_1) = \mathsf{sketchR}(\mathcal{J}_2)$*, resp.). Then there is a fractional matching from $\mathcal{J}_1$ to $\mathcal{J}_2$ such that every job $J \in \mathcal{J}_1$ is matched to an extent exactly 1 and every job $J \in \mathcal{J}_2$ is matched to an extent at most $1+\delta$. Moreover, if $J \in \mathcal{J}_1$ is matched to $J' \in \mathcal{J}_2$, then $d_J \geq d_{J'}$ ($r_J \leq r_{J'}$, resp.).*

Proof. We only focus on the case where jobs in $\mathcal{J}_1 \cup \mathcal{J}_2$ have the same releasing time. The case where they have the same deadline is analogous. Consider the following fractional bipartite-matching problem. The bipartite graph we are considering is $(\mathcal{J}_1, \mathcal{J}_2, E)$, where there is an edge $(J_1, J_2) \in E$ between a job $J_1 \in \mathcal{J}_1$ and a job $J_2 \in \mathcal{J}_2$ if the $d_{J_1} \geq d_{J_2}$.

By Tutte's theorem, it suffices to prove that for any non-empty subset $\mathcal{J}'_1$ of $\mathcal{J}_1$, we have $|\mathcal{J}_2(\mathcal{J}'_1)| \geq |\mathcal{J}'_1|/(1+\delta)$, where $\mathcal{J}_2(\mathcal{J}'_1)$ is the set of neighbors of $\mathcal{J}'_1$ in the bipartite graph. Focus on a subset $\mathcal{J}'_1 \subseteq \mathcal{J}_1$ and let t be the latest deadline of jobs in $\mathcal{J}'_1$. Suppose $\mathsf{sketchL}(\mathcal{J}_1) = \mathsf{sketchL}(\mathcal{J}_2) = (t_1, t_2, \cdots, t_\ell)$ and $t_j \leq t < t_{j+1}$ (assume $t_{\ell+1} = \infty$). Then we have $|\mathcal{J}'_1| \leq \Delta_{j+1} - 1$ since there are at most $\Delta_{j+1} - 1$ jobs in $\mathcal{J}_1$ with deadline before t_{j+1}. On the other hand, we have $|\mathcal{J}_2(\mathcal{J}'_1)| \geq \Delta_j$ since all jobs in $\mathcal{J}_2$ with deadlines before or at t_j are in $\mathcal{J}_2(\mathcal{J}'_1)$ and there are at least Δ_j such jobs. Notice that we have $\Delta_{j+1} \leq (1+\delta)\Delta_j + 1$. Thus, $|\mathcal{J}_2(\mathcal{J}'_1)| \geq \Delta_j \geq (\Delta_{j+1} - 1)/(1+\delta) \geq |\mathcal{J}'_1|/(1+\delta)$.

Let $\phi^{\mathsf{L}}_{i,j} = \mathsf{sketchL}(\mathcal{J}^{\mathsf{L}}_{i,j})$ and $\phi^{\mathsf{R}}_{i,j} = \mathsf{sketchR}(\mathcal{J}^{\mathsf{R}}_{i,j})$. Let $\phi = \{\phi^o_{i,j}\}_{o \in \{\mathsf{L},\mathsf{R}\}, i \in [q], j \in [z]}$(recall that z is the number of different weights). Instead of giving the set $\mathcal{J}_{\mathsf{up}}$ as input, we only give the sketch vector ϕ.

Thus, the approximate input of an extended WThr instance contains a block (A, B), a set $\mathcal{T}_{\mathsf{up}}$ of intervals and the sketch vector ϕ that approximates $\mathcal{J}_{\mathsf{up}}$. To ensure our sketch is a valid relaxation, we assume the best situation that $\mathcal{J}_{\mathsf{up}}$ is the "easiest" to schedule set of jobs that match the sketch vector ϕ. Intuitively, since left-side (right-side, resp.) jobs share the same release time A (deadline B, resp.), they become easier when they have later deadlines (earlier release times, resp.). We simply make the deadlines (release times, resp.) as late (early, resp.) as possible; meanwhile, we make the number of jobs as small as possible. The set is formally defined and returned by the procedure easiest-set in Algorithm 1. It is trivial that $\mathsf{cong}(\mathsf{easiest\text{-}set}(A, B, \phi), \mathcal{T}) \leq \mathsf{cong}(\mathcal{J}_{\mathsf{up}}, \mathcal{T})$ for any set $\mathcal{T}$ of scheduling intervals. This newly constructed easy set, $\mathsf{easiest\text{-}set}(A, B, \phi)$ replaces $\mathcal{J}_{\mathsf{up}}$. The number of possible inputs to a sub-instance is now affordable.

```
1: $\mathcal{J}' \leftarrow \emptyset$;
2: for each $(i, j) \in [q] \times [z]$ do
3:     $(t_1, t_2, \cdots, t_\ell) \leftarrow \phi^{\mathsf{L}}_{i,j}$;
4:     for $k \leftarrow 1$ to $\ell$ do
5:         add to $\mathcal{J}'$, $\Delta_k$ - $\Delta_{k-1}$ jobs of type $i$, with release time $A$, deadline $t_k$ and weight type $j$;
6:     $(t_1, t_2, \cdots, t_\ell) \leftarrow \phi^{\mathsf{R}}_{i,j}$;
7:     for $k \leftarrow 1$ to $\ell$ do
8:         add to $\mathcal{J}'$, $\Delta_k$ - $\Delta_{k-1}$ jobs of type $i$, with release time $t_k$, deadline $B$ and weight type $j$;
9: return $\mathcal{J}'$.
```

Algorithm 1: $\mathsf{easiest\text{-}set}(A, B, \phi)$: returning the "easiest" set of jobs agreeing with the sketches

PROPOSITION 3.1. *The total number of possible inputs to a sub-instance $(A, B, \mathcal{T}_{\mathsf{up}}, \phi)$ is $N^{O\left(\frac{\log n \log N \log W}{\epsilon^2 \delta}\right)}$.*

Proof. The total length of sketches in ϕ is at most $O(qz \log_{1+\delta} n)$. Since there are $O(N)$ different release times and deadlines, the number of possibilities for ϕ is $N^{O(qz \log_{1+\delta} n)} = N^{O\left(\frac{\log n \log N \log W}{\epsilon^2 \delta}\right)}$. This dominates the number of different $\mathcal{T}_{\mathsf{up}}$.

3.3 Wrapping Up The algorithm for processing each dynamic programming state is given in Algorithm 2. The output $f = f(A, B, \mathcal{T}_{\mathsf{up}}, \phi)$ is obtained with the relaxed "easiest" set $\mathcal{J}_{\mathsf{up}}$ that matches ϕ. Since this set is the easiest, it can be only larger than the actual maximum weight of jobs that can be scheduled in $\mathcal{J}_{\mathsf{in}}$ with the original ϕ. Our scheduling intervals F may not lead to a schedule of value f due to the relaxation performed in the sketches – however, we will show that this can be taken care of by increasing the congestion parameter c slightly.

From Line 2 to Line 8, we deal with the base case where (A, B) is a unit interval. When (A, B) has length at least 2, we enumerate all achievable pairs of inputs, $\left(A, C, \mathcal{T}^{\mathsf{L}}_{\mathsf{up}}, \phi^{\mathsf{L}}\right)$ and $\left(C, B, \mathcal{T}^{\mathsf{R}}_{\mathsf{up}}, \phi^{\mathsf{R}}\right)$, for the two sub-instances in Line 10. In Line 11, we find the decision function D for $\mathcal{J}_{\mathsf{up}} \cup \mathcal{J}_{\mathsf{in}}$ to achieve the combination. This involves some obvious comparisons such as whether $\mathcal{T}^{\mathsf{L}}_{\mathsf{up}}$ and $\mathcal{T}^{\mathsf{R}}_{\mathsf{up}}$ have the set of scheduling intervals intersecting time C. The only interesting part of this consistency check is avoiding enumerating all possible decision functions D for $\mathcal{J}_{\mathsf{in}} \cup \mathcal{J}_{\mathsf{up}}$, which was observed to be the main bottleneck for the naïve recursion. We show that the problem of finding the decision function D can be reduced to a bipartite graph matching problem and thus can be solved efficiently.

LEMMA 3.6. *Given $\mathcal{T}^{\mathsf{L}}_{\mathsf{up}}, \phi^{\mathsf{L}}$, and $\mathcal{T}^{\mathsf{R}}_{\mathsf{up}}, \phi^{\mathsf{R}}$, one can decide whether some valid decision function D achieves the combination in polynomial time.*

```
Input: $A, B, \mathcal{J}_{\mathsf{in}}$ defined by $(A,B)$, $\mathcal{T}_{\mathsf{up}}, \phi$;
Output: $f(A,B,\mathcal{T}_{\mathsf{up}},\phi)$: maximum weight of jobs in $\mathcal{J}_{\mathsf{in}}$ that can be scheduled;
$F(A,B,\mathcal{T}_{\mathsf{up}},\phi)$: set of job intervals hosting $\mathcal{J}_{\mathsf{in}}$ and $\mathcal{J}_{\mathsf{up}}$(defined in Line 1)
1:  $\mathcal{J}_{\mathsf{up}} \leftarrow \textsf{easiest-set}(A,B,\phi)$;
2:  if $B = A+1$ then
3:      if $|\mathcal{T}_{\mathsf{up}}| + |\mathcal{J}_{\mathsf{up}}| \le m$ and $(A,B)$ is a permissible interval for all jobs in $\mathcal{J}_{\mathsf{up}}$ then
4:          Select the $m - |\mathcal{T}_{\mathsf{up}}| - |\mathcal{J}_{\mathsf{up}}|$ heaviest jobs in $\mathcal{J}_{\mathsf{in}}$ (or the whole set $\mathcal{J}_{\mathsf{in}}$ if possible);
5:          $f(A,B,\mathcal{T}_{\mathsf{up}},\phi) \leftarrow$ total weight of the selected jobs;
6:          $F(A,B,\mathcal{T}_{\mathsf{up}},\phi) \leftarrow$ multi-set of $\min\{m - |\mathcal{T}_{\mathsf{up}}|, |\mathcal{J}_{\mathsf{up}}| + |\mathcal{J}_{\mathsf{in}}|\}$ intervals $(A,B)$;
7:      else $f(A,B,\mathcal{T}_{\mathsf{up}},\phi) \leftarrow -\infty$;
8:      return
9:  $C \leftarrow \lfloor (A+B)/2 \rfloor$, $f(A,B,\mathcal{T}_{\mathsf{up}},\phi) \leftarrow -\infty$, $\mathcal{J}'_{\mathsf{in}} \leftarrow \{J \in \mathcal{J}_{\mathsf{in}} : r_J < C < d_J\}$;
10: for every achievable pair of inputs $(A,C,\mathcal{T}^{\mathsf{L}}_{\mathsf{up}},\phi^{\mathsf{L}})$ and $(C,B,\mathcal{T}^{\mathsf{R}}_{\mathsf{up}},\phi^{\mathsf{R}})$ for the two sub-instances do
11:     $D \leftarrow$ valid decision function for $\mathcal{J}_{\mathsf{in}} \cup \mathcal{J}_{\mathsf{up}}$ achieving the two inputs;
12:     $f' \leftarrow f\left(A,C,\mathcal{T}^{\mathsf{L}}_{\mathsf{up}},\phi^{\mathsf{L}}\right) + f\left(C,B,\mathcal{T}^{\mathsf{R}}_{\mathsf{up}},\phi^{\mathsf{R}}\right) + \sum_{J \in \mathcal{J}'_{\mathsf{in}}, D(J) \ne \perp} w_J$;
13:     if $f' > f(A,B,\mathcal{T}_{\mathsf{up}},\phi)$ then
14:         $f(A,B,\mathcal{T}_{\mathsf{up}},\phi) \leftarrow f'$;
15:         $F(A,B,\mathcal{T}_{\mathsf{up}},\phi) \leftarrow F\left(A,C,\mathcal{T}^{\mathsf{L}}_{\mathsf{up}},\phi^{\mathsf{L}}\right) \uplus F\left(C,B,\mathcal{T}^{\mathsf{R}}_{\mathsf{up}},\phi^{\mathsf{R}}\right) \uplus \{D(J)\}_{J \in \mathcal{J}_{\mathsf{in}} \cup \mathcal{J}_{\mathsf{up}}, D(J) \notin \{\mathsf{L},\mathsf{R},\perp\}}$;
```

Algorithm 2: Processing a dynamic programming state

Proof. Let $\mathcal{J}'_{\mathsf{in}} \subseteq \mathcal{J}_{\mathsf{in}}$ be the set of jobs in $J \in \mathcal{J}_{\mathsf{in}}$ whose window (r_J, d_J) contains C. The decisions for jobs in $\mathcal{J}'_{\mathsf{in}} \cup \mathcal{J}_{\mathsf{up}}$ will affect the inputs to the two sub-instances. We claim checking whether some valid decision function for $\mathcal{J}'_{\mathsf{in}} \cup \mathcal{J}_{\mathsf{up}}$ achieves the combination of inputs can be reduced to a bipartite-graph matching problem.

The left side of the bipartite graph is the set of all jobs in $\mathcal{J}'_{\mathsf{in}} \cup \mathcal{J}_{\mathsf{up}}$. We need to find a matching such that each job in $\mathcal{J}_{\mathsf{up}}$ is matched exactly once and each job in $\mathcal{J}'_{\mathsf{in}}$ is matched at most once. The right-side vertex matched by a left-side job corresponds to the decision for the job. We shall add vertices to the right-side and edges to the graph.

Some trivial conditions on $\mathcal{T}^{\mathsf{L}}_{\mathsf{up}}, \mathcal{T}^{\mathsf{R}}_{\mathsf{up}}$ and $\mathcal{T}_{\mathsf{up}}$ must hold; otherwise, we claim there are no decision functions D to give the pair of inputs. From $\mathcal{T}^{\mathsf{L}}_{\mathsf{up}} \cup \mathcal{T}^{\mathsf{R}}_{\mathsf{up}} \setminus \mathcal{T}_{\mathsf{up}}$, we can determine the newly introduced aligned intervals (they must all contain C). For each aligned interval, we add a vertex to the right-side of the graph. There is an edge between a left-side job and a right-side interval if the interval is a permissible interval for the job. The newly added right-side vertices must be matched exactly once. Matching a job to an interval corresponds to scheduling the job in the interval.

We can also assign L or R to jobs in $\mathcal{J}'_{\mathsf{in}} \cup \mathcal{J}_{\mathsf{up}}$. Since the decisions are independent for different job types and weight types, we can focus on each job type-i and weight-type j. Let $\mathcal{J}'$ be the set of jobs of size-type i and weight type j in $\mathcal{J}'_{\mathsf{in}} \cup \mathcal{J}_{\mathsf{up}}$. Suppose $\phi^{\mathsf{R,L}}_{i,j} = (t_1, t_2, \cdots, t_\ell)$. Then for every $k \in [\ell]$, we have a right-side vertex representing the point t_k, and a right-side vertex representing the interval (t_k, t_{k+1}) (assuming $t_{\ell+1} = \infty$). Similarly we have right-side vertices for $\phi^{\mathsf{L,L}}_{i,j}$, $\phi^{\mathsf{L,R}}_{i,j}$ and $\phi^{\mathsf{R,R}}_{i,j}$. Consider a job $J \in \mathcal{J}'$ such that setting $D(J) = \mathsf{R}$ will make J a left-side job in the right-sub-block. If $d_J \in (t_k, t_{k+1})$ for some $k \in [\ell]$, then we connect J to the vertex on the right side for $\phi^{\mathsf{R,L}}_{i,j}$ representing the interval (t_k, t_{k+1}). If $d_J = t_k$ for some $k \in [\ell]$, then J is connected to 3 vertices for $\phi^{\mathsf{R,L}}_{i,j}$: the one representing the interval (t_{k-1}, t_k), the one representing the point t_k and the one representing the interval (t_k, t_{k+1}) (with the exception of $k = 1$, in which case J is only connected to 2 vertices). Similarly, we add edges by considering the case $D(J) = \mathsf{L}$. Each vertex on the right must be matched some suitable number of times. For example, the vertex for $\phi^{\mathsf{RL}}_{i,j}$ representing t_k needs to be matched exactly once for every $k \in [\ell]$, and the vertex representing (t_k, t_{k+1}) needs to be matched exactly $\Delta_{k+1} - \Delta_k - 1$ times for $k \in [\ell - 1]$ and at most $\Delta_{k+1} - \Delta_k - 1$ times for $k = \ell$.

Clearly, this bipartite matching problem can be solved efficiently, hence the lemma follows.

The final algorithm is as follows. For each block (A, B) in the recursion tree from bottom to top, we generate all possible $\mathcal{T}_{\mathsf{up}}, \phi$ and run Algorithm 2 for the input $A, B, \mathcal{T}_{\mathsf{up}}, \phi$. The most time-consuming part is enumerating $\left(A,C,\mathcal{T}^{\mathsf{L}}_{\mathsf{up}},\phi^{\mathsf{L}}\right)$ and $\left(C,B,\mathcal{T}^{\mathsf{R}}_{\mathsf{up}},\phi^{\mathsf{R}}\right)$ and verifying that they form a consistent combination. From Proposition 3.1 and Lemma 3.6, we upper bound the running time by $N^{O\left(\frac{\log n \log N \log W}{\epsilon^2 \delta}\right)}$.

We observe that our solution builds on a valid relaxation. Thus, $f(0, N, \emptyset, \phi) \ge \mathsf{opt}$, where the sketch $\phi^{\mathsf{L}}_{i,j}$ and $\phi^{\mathsf{R}}_{i,j}$ are all empty sequences. The correspondent solution signature returned is $\mathcal{T}^* = F(0, N, \emptyset, \phi)$. It is obvious from the algorithm that $\mathcal{T}^*$ can be allocated on m

machines.

We set $\delta = \Theta(\epsilon/\log N)$ so that $(1+\delta)^{\log N} \leq 1+\epsilon/3$. Then the running time of the algorithm is $n^{O\left(\frac{\log n \log^2 N \log W}{\epsilon^3}\right)}$, which is $2^{\mathrm{poly}(\log n, 1/\epsilon)}$ when N and W are $\mathrm{poly}(n)$. The following lemma yields Lemma 3.2 for this case. The proof uses Lemma 3.5 over $O(\log N)$ levels, thereby giving congestion $(1+\delta)^{\log N}$.

LEMMA 3.7. $\mathsf{MWM}(\mathcal{J}, \mathcal{T}^*, (1+\delta)^{\log N}) \geq \mathsf{opt}$.

Proof. For the proof, we shall use a slightly generalized definition of MWM. In this new definition, we have four parameters $\mathcal{J}'', \mathcal{J}', \mathcal{T}'$ and c, where $\mathcal{J}''$ and $\mathcal{J}'$ are two disjoint sets of jobs. The bipartite graph we are considering is $(\mathcal{J}'' \cup \mathcal{J}', \mathcal{T}', E)$. There is an edge $(J, T) \in E$ if T is a permissible interval of T. In the matching, every job in $\mathcal{J}''$ must be matched to an extent *exactly* 1, every job in $\mathcal{J}'$ must be matched to an extent at most 1 and every interval in $\mathcal{T}'$ must be matched to an extent at most c. The goal is to maximize the total weight of matched jobs in $\mathcal{J}'$, i.e, $\sum_{J \in \mathcal{J}'} w_J \sum_{T:(J,T)\in E} x_{(J,T)}$. Notice that we do not count the weight of matched jobs in $\mathcal{J}''$. Let $\mathsf{MWM}(\mathcal{J}'', \mathcal{J}', \mathcal{T}', c)$ be the value of the matching problem. Thus, the original $\mathsf{MWM}(\mathcal{J}', \mathcal{T}', c)$ is equivalent to $\mathsf{MWM}(\emptyset, \mathcal{J}', \mathcal{T}', c)$ in the new definition. If the problem is infeasible, we let the value be $-\infty$.

Focus on a block (A, B) in the tree. We say a block (A, B) is at level h if it has distance h to its nearest leaf. Thus, leaves are at level 0 and the root is at level $\log N$.

Focus on any level-h block (A, B) and any input $(A, B, \mathcal{T}_{\mathsf{up}}, \phi)$. Let $\mathcal{J}_{\mathsf{in}} = \{J \in \mathcal{J} : A \leq r_J < d_J \leq B\}$ be the jobs whose windows are in (A, B). Let $\mathcal{J}_{\mathsf{up}}$ be the set returned by $\mathsf{easiest-set}(A, B, \phi)$. We show that for *any* set $\mathcal{J}'_{\mathsf{up}}$ of jobs that matches the sketch vector ϕ, the returned set $\mathcal{T} = F(A, B, \mathcal{T}_{\mathsf{up}}, \phi)$ satisfies $\mathsf{MWM}(\mathcal{J}'_{\mathsf{up}}, \mathcal{J}_{\mathsf{in}}, \mathcal{T}, (1+\delta)^h) \geq \mathsf{opt}'$, where opt' is the optimal value for the extended WThr instance defined by (A, B), $\mathcal{T}_{\mathsf{up}}$ and $\mathcal{J}'_{\mathsf{up}}$.

We prove the statement by induction on h. For the base case $h = 0$, this is obviously true since the only choice for $\mathcal{J}'_{\mathsf{up}}$ is $\mathcal{J}_{\mathsf{up}}$ and our algorithm is optimal for this case. Focus on the case $h \geq 1$. Let opt be the maximum weight for the extended WThr problem defined by (A, B), $\mathcal{T}_{\mathsf{up}}$ and $\mathcal{J}_{\mathsf{up}}$, and $\mathcal{T}$. From the induction hypothesis, it is easy to see that $\mathsf{MWM}(\mathcal{J}_{\mathsf{up}}, \mathcal{J}_{\mathsf{in}}, \mathcal{T}, (1+\delta)^{h-1}) \geq \mathsf{opt}'$. By applying Lemma 3.5 to each combination of job-type i, weight-type j and side-type (left-side or right-side jobs), it is easy to see that we can find a fractional matching x from $\mathcal{J}'_{\mathsf{up}}$ to $\mathcal{J}_{\mathsf{up}}$ such that every job in $\mathcal{J}'_{\mathsf{up}}$ is matched to an extent exactly 1 and every job in $\mathcal{J}_{\mathsf{up}}$ is matched to an extent at most $1+\delta$. Moreover, if $J' \in \mathcal{J}'_{\mathsf{up}}$ is matched to $J \in \mathcal{J}_{\mathsf{up}}$, then every permissible interval for J is also a permissible interval for J'. Consider the fractional matching x' achieving $\mathsf{MWM}(\mathcal{J}_{\mathsf{up}}, \mathcal{J}_{\mathsf{in}}, \mathcal{T}, (1+\delta)^{h-1})$. Recall that in x' all jobs in $\mathcal{J}_{\mathsf{up}}$ are matched to an extent of exactly 1. Combining matchings x and x', we can obtain a fractional matching x'' between $\mathcal{J}'_{\mathsf{up}} \cup \mathcal{J}_{\mathsf{in}}$ and $\mathcal{T}$ where every job in $\mathcal{J}'_{\mathsf{up}}$ is matched to an extent exactly 1, every job in $\mathcal{J}'_{\mathsf{in}}$ is matched to an extent at most 1 and every job in $\mathcal{T}$ is matched to an extent at most $(1+\delta)^h$. The extent to which a job $J \in \mathcal{J}_{\mathsf{in}}$ is matched in x' is the same as that in x''. Thus, $\mathsf{MWM}(\mathcal{J}'_{\mathsf{up}}, \mathcal{J}_{\mathsf{in}}, \mathcal{T}, (1+\delta)^h) \geq \mathsf{MWM}(\mathcal{J}_{\mathsf{up}}, \mathcal{J}_{\mathsf{in}}, \mathcal{T}, (1+\delta)^{h-1}) \geq \mathsf{opt}'$.

4 Dealing with Large N and W for Machine Minimization and Throughput

4.1 Sketch of Algorithms We first sketch how to prove Lemma 3.2 when N and W are large. Basically there are two barriers: (1) the number of blocks (A, B) in the recursion tree can be exponential; (2) the number of different job types (size/weight) can be $\omega(\mathrm{poly}\log n)$.

We now show how to remove the first barrier. We distinguish *flexible* jobs from non-flexible jobs: flexible jobs are the jobs J satisfying $(d_J - r_J)/p_J \geq \mathrm{poly}(n, 1/\epsilon)$ for some suitable function $\mathrm{poly}(n, 1/\epsilon)$. Intuitively, these jobs J are flexible in the following sense: if there is a big job scheduled in J's window, then with additional speed augmentation, J can be scheduled with the big job; otherwise, J has plenty of empty space within its window and can be flexibly scheduled. Using this intuition, we show that by using $(1+\epsilon)$-speed, we can assume all flexible jobs have size 0. Notice that in the non-preemptive setting, jobs of size 0 make sense. Then, each job either has size 0 or size $p_J \geq (d_J - r_J)/\mathrm{poly}(n, 1/\epsilon)$. With this property, we can then identify a set of interesting points: the points in which a job starts or ends. Using the above property and aligned intervals, we can bound the number of interesting points by $\mathrm{poly}(n, 1/\epsilon)$. In our DP, we can focus on the blocks (A, B) where both A and B are interesting points. This will reduce the number of blocks to $\mathrm{poly}(n, 1/\epsilon)$.

Now consider the second barrier. We can wlog. assume W is polynomial since we can remove jobs of weight at most $\epsilon W/n$; if jobs are unweighted, there is nothing to do here. By scaling, we can assume W is $O(n/\epsilon)$. The only thing that remains is to handle the large number of different job sizes. The key idea is that when we divide a block (A, B) into two blocks (A, C) and (C, B), we choose a point C that does not belong to the window (r_J, d_J) of a small job J. A job J is small if $0 < p_J \leq (B - A)/\mathrm{poly}(n, 1/\epsilon)$. If the quantity $\mathrm{poly}(n, 1/\epsilon)$ is suitable, we can still choose a C that almost equally splits (A, B) (recall that if $p_J > 0$, then $d_J - r_J \leq p_J \cdot \mathrm{poly}(n, 1/\epsilon)$). Then, the two sub-blocks are (A, C') and (C'', B), where C' and C'' are the interesting points to the left and the right of C respectively. Consider the set $\mathcal{J}_{\mathsf{up}}$ of jobs for a fixed (A, B). The size of a job $J \in \mathcal{J}_{\mathsf{up}}$ is at most $B - A$

since otherwise it can not be scheduled in (A, B). Also, it is at least $(B - A)/\mathrm{poly}(n, 1/\epsilon)$. This holds because of the way blocks are cut: if a job J has very small positive size (thus, very small window) compared to $B - A$, then we avoided cutting its window in upper levels. Thus, for such a job J, (r_J, d_J) is either contained in (A, B) or disjoint from (A, B). Therefore, there are only $O(\log n)$ different job types in $\mathcal{J}_{\mathsf{up}}$ and we can afford to use our sketching scheme for $\mathcal{J}_{\mathsf{up}}$. It is true that our recursion tree can have height much larger than $O(\log n)$. However, for each block (A, B), the congestion factor we lose is only on the job-types which are present in $\mathcal{J}_{\mathsf{up}}$. By using a careful type-wise analysis for the congestion, we can bound the overall congestion by $(1 + \delta)^{O(\log n)}$.

4.2 Detailed Algorithms We now give detailed algorithms to deal with large N and W. For WThr, we can assume W is at most $O(n/\epsilon)$, only losing a $1 - \epsilon/4$ factor in the approximation ratio. Consider the largest weight W. If some job has weight at most $\epsilon W/(4n)$, then we can discard it. Since our optimal value is at least W and we discard at most $\epsilon W/4$ total weight, we only lose a factor of $1 - \epsilon/4$ in the approximation ratio. By scaling, we can assume all weights are positive integers less than $O(n/\epsilon)$.

In the first step, we shall deal with what we call *flexible* jobs. These are the jobs J with p_J much smaller than $d_J - r_J$.

DEFINITION 4.1. (FLEXIBLE JOBS) *A job $J \in \mathcal{J}$ is flexible if $p_J \le \frac{\epsilon(d_J - r_J)}{6n^2}$.*

We show that with some small speed, we assume the sizes of flexible jobs are 0. Notice that in non-preemptive scheduling, jobs of size 0 make sense.

LEMMA 4.2. *With $(1 + \epsilon/4)$-speed, we can assume flexible jobs have size 0.*

Proof. We change the sizes of flexible jobs to 0 and then find a scheduling of $\mathcal{J}$ on m machines. Then, we change the size of flexible jobs back to their original sizes.

Fix some machine and some job J of size 0 scheduled on the machine. We change its size back to its original size p_J. We say a job J' is small if its current size (which is either 0 or $p_{J'}$) is at most $5np_J/\epsilon$ and big otherwise. Consider the case where no big jobs scheduled on the machine intersect $(r_J + p_J, d_J - p_J)$. Then, since $(d_J - r_J) \ge 6n^2 p_J/\epsilon$ and there are at most n jobs, we can find a free segment of length at least $(6n^2 p_J/\epsilon - 2p_J - 5np_J/\epsilon \times n)/n \ge p_J$ in $(d_J + p_J, r_J - p_J)$ on the machine. Thus, we can schedule the job J in the free segment. Now consider the case where some big job J' scheduled on the machine intersects $(r_J + p_J, d_j - p_J)$. Since the 0-sized job J was scheduled somewhere in (r_J, d_J), the scheduling interval for J' can not cover the whole window (r_J, d_J). Then, we can either cut the first or the last p_J time slots of scheduling interval for J' and use them to schedule J. The length of the scheduling interval for J' is reduced by $p_J \le \epsilon p_{J'}/(5n)$. Thus, we have changed the size of J back to its original size p_J.

For each size J', the scheduling interval for J' is cut by at most $\epsilon p_{J'}/(5n) \times n = \epsilon p_{J'}/5$. The new scheduling is valid if the machines have speed $1 + \epsilon/4 \ge 1/(1 - \epsilon/5)$.

With the sizes of flexible jobs changed to 0, we then define the aligned intervals of positive size and define permissible intervals as in Section 2.1. We say a point in the time horizon $(0, N)$ is an interesting point if it is the starting-point or the ending-point of some permissible interval for some positive-sized job, or if it is r_J for some 0-sized job J.

LEMMA 4.3. *The total number of interesting points is $O(n^3/\epsilon^2)$.*

Proof. Focus on a positive job J. Suppose it is of type i, defined by (s_i, g_i). Then the number of permissible intervals for J is $\lceil (d_J - r_J)/g_i \rceil \le O(n^2 s_i/(\epsilon g_i)) = O(n^2/\epsilon^2)$, where the last equation is by Lemma 2.1. Since there are n jobs, the total number of interesting points is bounded by $O(n^3/\epsilon^2)$.

We can assume that jobs start and end at interesting points. This is clearly true for positive-sized jobs. For a 0-sized job, we can move them to left until it hits its arrival time or ending-point of some other interval. Our algorithm is still based on the naive dynamic programming. In order to improve the running time, we need to make two slight modifications to the naive dynamic programming.

First, when we are solving an extended WThr instance on block (A, B), we change (A, B) to (A', B'), where A' is the interesting point to the right of A and B' interesting point to the left of B. By our definition of interesting points, this does not change the instance. The base cases are when (A, B) contains no interesting points. With this modification, the tree of blocks defined by the recursive algorithm can only contain $O(n^3/\epsilon^2)$ blocks.

Second, we change the way we choose C. For a fixed (A, B), we say a positive-sized job J is small if its size is at most $\epsilon(B - A)/(12n^3)$ and big otherwise. We select our C such that C is not contained in any window (r_J, d_J) of small jobs J. Since all positive-sized jobs are non-flexible, the total window size over all small jobs J is at most $n \times \epsilon(B - A)/(12n^3) \times 6n^2/\epsilon \le (B - A)/2$. Thus, we can choose C such that $A + (B - A)/4 \le C \le A + 3(B - A)/4$.

With the two modifications, we can describe our sketching scheme. Fix a block (A, B). It is clear that A and B will not be contained in window (r_J, d_J) for any small job J. This is true since in small jobs w.r.t (A, B)

are also small w.r.t super-blocks of (A, B). Then when cutting blocks in upper levels, we avoid cutting windows of small jobs. If a job has size greater than $B - A$, then it clearly can not be scheduled in (A, B). Thus, for fixed (A, B), $\mathcal{J}_{\mathsf{up}}$ will only contain big jobs of size at most $B - A$. That is, $\mathcal{J}_{\mathsf{up}}$ only contain jobs of size more than $\epsilon(B - A)/(12n^3)$ and at most $B - A$. There can be at most $O(\log n/\epsilon)$ different job types in $\mathcal{J}_{\mathsf{up}}$ (jobs of size 0 are of the same type). Using the same sketching scheme as in Section 3.2, the total number of different sketches for fixed (A, B) can be at most $n^{O(\log n/\epsilon) z O(\log_{1+\delta} n)} = n^{O\left(\log^3 n/(\delta\epsilon)\right)}$.

We now analyze the congestion we need. It is true that the tree of blocks may have up to n levels. However, for a specific job type, the number of levels where we lose a factor in the congestion for the job type is $O(\log n)$. In order to prove this more formally, we need to generalize the definition of the maximum weighted matching. $\mathsf{MWM}(\mathcal{J}', \mathcal{J}'', \mathcal{T}', c)$ is defined the same as before, except that now $c : [q] \to \mathbb{R}$ a vector of length q. If an aligned interval $T \in \mathcal{T}'$ is of type $i \in [q]$, then it can be matched to an extent at most c_i.

For every block (A, B) in the recursion tree, we shall define a congestion vector $c : [q] \to \mathbb{R}$ such that the following holds. Consider any approximate input $(A, B, \mathcal{T}_{\mathsf{up}}, \phi)$ to an extended WThr instance on (A, B). Let $\mathcal{J}_{\mathsf{in}} = \{J \in \mathcal{J} : A \leq r_J < d_J \leq B\}$ be the jobs whose windows are in (A, B). Let $\mathcal{J}_{\mathsf{up}}$ be the set returned by $\mathsf{easiest-set}(A, B, \phi)$. Then for any set $\mathcal{J}'_{\mathsf{up}}$ of jobs that matches the sketch vector ϕ, the returned set $\mathcal{T} = F(A, B, \mathcal{T}_{\mathsf{up}}, \phi)$ satisfies $\mathsf{MWM}(\mathcal{J}'_{\mathsf{up}}, \mathcal{J}_{\mathsf{in}}, \mathcal{T}, c) \geq \mathsf{opt}'$, where opt' is the value for the extended WThr instance defined by (A, B), $\mathcal{T}_{\mathsf{up}}$ and $\mathcal{J}'_{\mathsf{up}}$.

It is obvious that we can set the c to be all-one vector for base blocks. Suppose have defined the congestion vector c^1, c^2 for the two sub-blocks of (A, B). We now define the congestion vector c for the block (A, B). From the proof of Lemma 3.7, it is easy to see that we can define c be the following vector. If $\epsilon(B - A)/(12n^3) \leq s_i \leq B - A$ (recall that s_i is the size of type-i jobs), we let $c_i = (1 + \delta) \max\left\{c_i^1, c_i^2\right\}$; otherwise, we let $c_i = \max\left\{c_i^1, c_i^2\right\}$.

Now consider the final c we get for the root block. Then $c_i = (1 + \delta)^h$, where h is the maximum number of blocks (A, B) in a root-to-leaf path of the recursion tree, such that $\epsilon(B - A)/(12n^3) \leq s_i \leq B - A$. Since a child block has size at most $3/4$ times its parent block, h is at most $O(\log(12n^3/\epsilon)) = O(\log n)$. Thus, by setting $\delta = \Omega(\epsilon/\log n)$, we finish the proof of Lemma 3.2 with running time $2^{\mathrm{poly}(\log n, 1/\epsilon)}$.

5 Algorithms for (Weighted) Flow Time

The algorithms for FT and WFT follow from the same framework that we used for WThr with only minor modifications; thus we only highlight the differences. In FT and WFT, jobs have no deadlines. Hence we start with an obvious upper bound on $N := \max_{J' \in \mathcal{J}} r_{J'} + \sum_{J' \in \mathcal{J}} p_{J'}$ – clearly, all jobs can be completed by time N in any optimal solution. Proposition 2.1 again holds for FT and WFT. The only difference is that we solve a minimum-cost perfect matching problem for the same bipartite graph used in the proof where each edge between J and T has a cost that is w_J times the flow time or tardiness of J when J is assigned in T. In the naïve DP, we do not have the option of discarding a job. Hence $D(J) = \perp$ is disallowed.

We now focus on WFT. One crucial observation for WFT is that jobs of the same type (size and weight) can be scheduled in First-In-First-Out manner since jobs do not have deadlines. Using this observation, we can obtain a simpler $(1 + \epsilon, 1 + \epsilon)$-approximation for WFT with polynomial N and W. The DP has a "path-structure" : it proceeds from time 0 to time N. However, to handle WFT with arbitrary N and W, we need to use our tree-structure DP framework. Hence we stick with our general framework.

We first consider the case when N and W are $\mathrm{poly}(n)$. In each sub-instance in the recursion, we are given a block (A, B), a set $\mathcal{T}_{\mathsf{up}}$ of already allocated intervals and a set $\mathcal{J}_{\mathsf{up}}$. Since jobs have deadlines ∞, there are no jobs in $\mathcal{J}_{\mathsf{in}}$. The goal is to schedule all jobs in $\mathcal{J}_{\mathsf{up}}$ inside (A, B) so as to minimize the total weighted flow time. With the above crucial observation, we can specify $\mathcal{J}_{\mathsf{up}}$ exactly. Focus on jobs of type (i, j) in $\mathcal{J}$. We order these jobs by ascending release times. With this observation, the set of type-(i, j) jobs in $\mathcal{J}_{\mathsf{up}}$ must appear consecutively in the ordering (assuming a consistent way to break ties). Thus, we only need to store two indices indicating the first job and the last job of each type. Since the DP is exact, we do not lose anything from it. The only factors we lose are from the pre-processing step: a $(1+\epsilon)$-speed due to rounding job sizes and aligning jobs, and a $(1 + \epsilon)$-approximation factor due to rounding weights. The second factor is unnecessary for FT. Thus, we obtain a $(1 + \epsilon, 1)$-approximation for FT and the $(1 + \epsilon, 1 + \epsilon)$-approximation for WFT when N and W are $\mathrm{poly}(n)$.

Now we show how to extend the above algorithm for WFT to the case of large N and W. The overall ideas are similar to those we used for MM, and WThr. We begin with an easy upper bound on the optimum objective opt for our problem.

Claim 5.1. $\sum_J w_J p_J \leq \mathsf{opt} \leq 2n^2 \max_J w_J p_J$.

Proof. The lower bound on opt is trivial; we focus on the upper bound. We schedule jobs in non-increasing order of weights and we say $J' < J$ if J' is before J in the ordering. If we try to schedule each job J as early as possible, then it is easy to see that job J's weighted flow time is at most $w_J(\sum_{J'<J} p_{J'} + np_J) \leq \sum_{J' \leq J} w_{J'}p_{J'} + w_J np_J$. Summing this upper bound over all jobs yields the lemma.

The next step is to simplify "negligible" jobs. We say that a job J is negligible if $w_J p_J \leq \frac{\epsilon^2}{8n^5}$ opt, otherwise non-negligible. As usual, we can guess opt using a binary search.

LEMMA 5.2. *With $(1+2\epsilon)$ extra speed, we can assume that negligible jobs have size 0. More precisely, making some jobs have size 0 can only decrease the total weighted flow time to schedule all jobs, and with $(1+2\epsilon)$ extra speed, we can in polynomial time convert a schedule for the simplified instance to a schedule for the original instance without increasing the total weighted flow time.*

Proof. Clearly, making some jobs have size 0 can only decrease the total weighted flow time to schedule all jobs. To show the second part of the claim, we consider negligible jobs J in an arbitrary order, and revert their current sizes 0 to their respective original sizes p_J. For each negligible job J, we push back the job to the right until we find either an empty space of size p_J or a job J' of size np_J/ϵ – this job J' can be either a non-negligible job, or a negligible job whose job size has been reverted to its original size. Note that job J can move at most $n(p_J + np_J/\epsilon)$ time steps. Hence this will increase the objective by at most $\frac{2n^2}{\epsilon} w_J p_J \leq \frac{\epsilon}{4n^3}$ opt $\leq \frac{\epsilon}{2n} \max_J w_J p_J$. Hence the total increase of the objective due to moving negligible jobs is at most $\frac{\epsilon}{2} \max_J w_J p_J$. This loss can be offset by shrinking the job J that maximizes $w_J p_J$ by a factor of $(1-\epsilon/2)$. This can be done using a $(1+\epsilon)$-speed augmentation.

We now show that we can still get a feasible schedule with a small amount of extra speed augmentation. If job J found an empty space to schedule itself, there's nothing we need to do. If job J found a big job J' with size at least np_J/ϵ, then we shrink job J' using speed augmentation to make a minimal room for J. In the worse case, job J' can be shrink by a factor of $(1-\epsilon/n)^{n-1} \geq 1-\epsilon$ by all other jobs. Thus $1/(1-\epsilon)$ extra speed augmentation is enough to get a feasible schedule for the original instance. The lemma follows by observing $1/(1-\epsilon) \leq 1+2\epsilon$ for a sufficiently small $\epsilon > 0$, and rescaling ϵ appropriately.

For each job J, we define a "fictitious" but safe deadline d_J such that $w_J(d_J - r_J) \leq$ opt. We make a simple observation which follows from the definition of negligible jobs and deadlines.

PROPOSITION 5.1. *For all negligible jobs J, $p_J \leq \frac{\epsilon^2}{4n^5}(d_J - r_J)$. For all non-negligible job J, $p_J \geq \frac{\epsilon^2}{8n^5}(d_J - r_J)$.*

We now use the same trick of avoid cutting small-window jobs as we did for MM and WThr. Consider a sub-instance on block (A, B). We say that job J has a small window (with respect to (A, B)) if $d_J - r_J \leq \frac{1}{n^2}(B - A)$; other jobs have a large window. As before, we can find a nearly middle point C of (A, B) without cutting any small-window jobs. Hence we focus on large-window jobs. Let's first consider large-window non-zero-sized jobs. All such jobs have size at least $\frac{\epsilon^2}{8n^7}(B - A)$. Knowing that only jobs of size at most $(B - A)$ are considered, we can conclude that the sizes of all large-window non-zero-sized jobs are within factor $O(n^7)$. Also from definition of non-negligible jobs, we know that the weights of all large-window non-zero-sized jobs are within factor $O(n^{12})$. Hence we only need to consider $O((\log^2 n)/\epsilon)$ different types (sizes and weights) of jobs.

Now let's focus on the 0-sized jobs. If jobs are unweighted, then these jobs are of the same type, hence we can get a 1-approximaiton – note that we do not need to round jobs weights, and we lost no approximation factor so far. However, if jobs have weights we need an additional step since the 0-sized jobs may have a wide range of weights. Note that we can assume that 0-sized jobs have large windows. Hence for all 0-sized jobs which can be potentially cut, we have $d_J - r_J \geq \frac{1}{n^2}(B - A)$. For a 0-sized job J such that $d_J - r_J \geq n^2(B - A)$, we reduce job J's weight to 0, and these jobs will be taken care of as an equal type: This is justified for the following reason. We know that at this point no matter where we schedule job J within (A, B), job J's weighted flow time can be affected by at most $w_J(B - A) \leq w_J(d_J - r_J)/n^2 \leq$ opt$/n^2$. Hence we now have that for all 0-sized non-zero-weight jobs J, $(B - A)/n^2 \leq d_J - r_J \leq n^2(B - A)$. This implies that the weights of 0-sized non-zero-weight jobs are now all within factor $O(n^4)$. Hence we only have $O((\log n)/\epsilon)$ different types (weight) of 0-sized jobs.

6 Algorithms for (Weighted) Tardiness

If we simply copy the algorithm for MM and WThr and hope it works for WTar, then we have a barrier: the set $\mathcal{J}_{\text{in}}$ is not determined by the block (A, B). In WTar, even if a job J has window (r_J, d_J) completely contained in (A, B), we may schedule J after B. Thus, we need to store the set $\mathcal{J}_{\text{in}}$. The sketching scheme used in MM and WThr utilizes a key property: the set of jobs for which we are going to construct a sketch either have the same release time or the same deadline. However, in $\mathcal{J}_{\text{in}}$, jobs can have different release times and different deadlines. Thus, the sketching scheme will not work.

To remove this barrier, we modify the naïve DP slightly. Suppose in some instance on a block (A', B'), we need to schedule some jobs J with $d_J \le A'$ of type-(i,j). Then, all these jobs can be treated equally – after paying cost $A' - d_J$ for each of them, we can treat them as having the same release time and deadline A'. We design our DP so that the tardiness $A' - d_J$ is already charged in some other sub-instance and this instance is not responsible for it. Thus, we only need to know how many such jobs J need to be scheduled.

With this observation in mind, let us focus on the block (A, B). If we have decided to schedule some job $J \in \mathcal{J}_{\mathsf{in}}$ with $A \le r_J \le d_J \le B$ to the right of B, then by the above argument, the sub-instances on blocks to the right of B do not care about the release time and deadline of J. They only care about their job types and weight types. Thus, in the instance for (A, B), it suffices to have a number $h_{i,j}$ in the input indicating the number of jobs $J \in \mathcal{J}_{\mathsf{in}}$ of type (i,j) that are scheduled to the right of B. In the sub-instance for (A, B), we just need to push the best set of jobs in $\mathcal{J}_{\mathsf{up}}$ to the right of B. The cost of pushing back jobs is included in the objective.

We remark that we need 4-speed in the preprocessing step to make permissible intervals form a laminar family. This is to avoid some involved case analysis. Thus the final speed is $8 + \epsilon$. We believe our ideas can lead to a $(2+\epsilon, 1+\epsilon)$-approximation by using a more complicated DP.

We now describe our algorithm for WTar in more details. We first assume N and W are polynomial. Assume N is a power of 2. Then we build a perfect binary tree of blocks of depth $\log N$, where the root block is $(0, N)$, the two child-blocks equally split the parent-block, and leaf-blocks are blocks of unit length. By using 4-speed, we can assume each job J has size 2^i for some $0 \le i \le \log N$ and a permissible interval of such a job is a block of length 2^i in the tree. Thus, the set of all permissible intervals form a laminar family. We say a job is of type-i if its size is 2^i. By losing a factor of $1+\epsilon$ in the approximation ratio, we can assume the number of different weights is at most $O((\log W)/\epsilon)$; we can index the weights using integers from 1 to $O((\log W)/\epsilon)$. Our algorithm is based on a slightly modified version of the naïve DP described in Section 2.3. We first describe the sub-instance in the recursion and explain the meaning of each term. The input to a sub-instance is as follows.

1. an aligned block (A, B);
2. a number m' indicating the number of allocated intervals containing (A, B);
3. $\mathcal{J}_{\mathsf{up}}$, a subset of jobs J in $\mathcal{J}$; for each $J \in \mathcal{J}_{\mathsf{up}}$, $(r_J, d_J) \cap (A, B) \ne \emptyset$ and $(r_J, d_J) \not\subset (A, B)$;
4. $g_{i,j}, i \le \log(B - A)$ and $j \in [z]$;
5. $h_{i,j}, i \le \log(B - A)$ and $j \in [z]$.

Block (A, B) and $\mathcal{J}_{\mathsf{up}}$ is as in the algorithm for WThr. We must schedule all jobs in $\mathcal{J}_{\mathsf{up}}$ in (A, B). m' is correspondent to $\mathcal{T}_{\mathsf{up}}$. Since our permissible intervals form a laminar family, and (A, B) is a permissible interval, it suffices to use a single number m'. By the definition of m', $m - m'$ machines are available during the interval (A, B).

We now explain $\{g_{i,j}\}_{i,j}$ and $\{h_{i,j}\}_{i,j}$. The input $\{g_{i,j}\}_{i,j}$ defines the set of jobs with deadlines before or at A that must be scheduled in the block (A, B). We use $\mathcal{J}_{\mathsf{L}}$ to denote these jobs. Then, for every integer $0 \le i \le \log(B - A)$ and $j \in [z]$, there are exactly $g_{i,j}$ such jobs of type (i, j). Notice the tardiness of each job $J \in \mathcal{J}_{\mathsf{L}}$ is at least $A - d_J$. We shall design our recursion so that the $A - d_J$ tardiness is already considered and the instance for (A, B) is not responsible for this cost. Thus, we treat jobs in $\mathcal{J}_{\mathsf{L}}$ as having arrival times and deadlines equaling A. We also assume $\mathcal{J}_{\mathsf{L}}$ is a newly created set that is disjoint from $\mathcal{J}$.

Let $\mathcal{J}_{\mathsf{in}} = \{J \in \mathcal{J} : (r_J, d_J) \in (A, B)\}$ as in the algorithm for WThr. Then $h_{i,j}$ is the number of type-(i,j) jobs in $\mathcal{J}_{\mathsf{in}}$ that need to be scheduled outside (A, B), i.e, in an interval with starting time greater than or equal to B. For such a job J, this instance is responsible for a tardiness of $B - d_J$. The remaining tardiness for J is counted in some other instance.

We can now describe the goal of our instance. We need to select a set $\mathcal{J}_{\mathsf{R}} \subseteq \mathcal{J}_{\mathsf{in}}$ of jobs, such that for every i, j, $\mathcal{J}_{\mathsf{R}}$ contains exactly $h_{i,j}$ type-(i,j) jobs. The goal is to schedule all jobs in set $\mathcal{J}_{\mathsf{L}} \cup \mathcal{J}_{\mathsf{up}} \cup (\mathcal{J}_{\mathsf{in}} \setminus \mathcal{J}_{\mathsf{R}})$ inside (A, B) in $m - m'$ machines so as to minimize the total weighted tardiness of jobs in $\mathcal{J}_{\mathsf{L}} \cup \mathcal{J}_{\mathsf{up}} \cup \mathcal{J}_{\mathsf{in}}$. For jobs in $\mathcal{J}_{\mathsf{L}} \cup \mathcal{J}_{\mathsf{up}} \cup (\mathcal{J}_{\mathsf{in}} \setminus \mathcal{J}_{\mathsf{R}})$, the tardiness is defined normally. For jobs $J \in \mathcal{J}_{\mathsf{R}}$, the tardiness is defined as $B - d_J$. As described earlier, the instance is responsible for the tardiness of jobs in $\mathcal{J}_{\mathsf{R}}$ up to time B.

We proceed with describing how to reduce the instance to two sub-instances on (A, C) and (C, B). Let's first make decisions for jobs of type $i = \log(B - A)$. We need to schedule these jobs in the entire block (A, B) or to the right of B. For a job $J \in \mathcal{J}_{\mathsf{up}} \cup \mathcal{J}_{\mathsf{L}}$ of type i, we must schedule it in (A, B) and incur a cost of $w_J \max\{0, B - d_J\}$. All jobs $J \in \mathcal{J}_{\mathsf{in}}$ of type $i = \log(B - A)$ must have the same arrival time A. For each weight type j, we shall add $h_{i,j}$ jobs of type-(i,j) in $\mathcal{J}_{\mathsf{in}}$ with the largest deadlines to $\mathcal{J}_{\mathsf{R}}$ and schedule the remaining jobs in (A, B). With jobs of type $i = \log(B - A)$ scheduled, the m' parameters for the two sub-instances are determined.

We need to define the other parameters: $\mathcal{J}^{\mathsf{L}}_{\mathsf{up}}, \{g^{\mathsf{L}}_{i,j}\}_{i,j}, \{h^{\mathsf{L}}_{i,j}\}_{i,j}$ for the instance on (A, C) and $\mathcal{J}^{\mathsf{R}}_{\mathsf{up}}$, $\{g^{\mathsf{R}}_{i,j}\}_{i,j}, \{h^{\mathsf{R}}_{i,j}\}_{i,j}$ for the instance on (C, B). We first initialize all integer parameters to 0 and set parameters to $\emptyset$.

For each job $J \in \mathcal{J}_{\mathsf{up}}$ of type $i < \log(B - A)$ and weight type j, we have two choices for J: either pass it to the left instance or the right instance. If (r_J, d_J) does not intersect (C, B) and we chose to pass J to the right instance, then we have to increase $g^{\mathsf{R}}_{i,j}$ by 1 and pay a cost of $w_J(C - d_J)$. If (r_J, d_J) does not intersect (A, C) then we can not pass J to the left instance. In other cases, we add J to $\mathcal{J}^{\mathsf{L}}_{\mathsf{up}}$ or $\mathcal{J}^{\mathsf{R}}_{\mathsf{up}}$ without incurring any cost.

It is easy to make decisions for jobs in $\mathcal{J}_{\mathsf{L}}$. For each job type $i < \log(B - A)$ and weight type j, we enumerate the number of type-(i, j) jobs that will be scheduled in (A, C) and the number of these jobs that will be scheduled in (C, B) (the two numbers sum up to $g_{i,j}$). We add the first number to $g^{\mathsf{L}}_{i,j}$ and the second number to $g^{\mathsf{R}}_{i,j}$. If we passed a job to in $\mathcal{J}_{\mathsf{L}}$ to the (C, B), we incur a cost of $w_J(C - A)$.

Now, we consider jobs in $\mathcal{J}_{\mathsf{in}}$. Fix some job type $i < \log(B - A)$ and weight type j. Each job $J \in \mathcal{J}_{\mathsf{in}}$ can fall into three categories depending on its window (r_J, d_J): (1) $(r_J, d_J) \subseteq (A, C)$; (2) $(r_J, d_J) \subseteq (C, B)$; (3) $r_J < C < d_J$. We enumerate the number of jobs in each category that will be pushed to $\mathcal{J}_{\mathsf{R}}$. Notice that these three numbers sum up to $h_{i,j}$. We add the first number to $h^{\mathsf{L}}_{i,j}$ and the second number to $h^{\mathsf{R}}_{i,j}$. We incur a cost of $w(B-C)$ times the first number, where w is the weight for weight type j. Also, we enumerate the number of jobs in category (1) that will be passed to the right instance. We add the number to $h^{\mathsf{L}}_{i,j}$ and to $g^{\mathsf{R}}_{i,j}$. We make individual choices for jobs in category (3). Each job J can either be added to $\mathcal{J}_{\mathsf{R}}$, in which case we incur a cost of $w_J(B-d_J)$, or passed to the left instance, in which case we add it to $\mathcal{J}^{\mathsf{L}}_{\mathsf{up}}$, or passed to the right instance, in which case we add it to $\mathcal{J}^{\mathsf{R}}_{\mathsf{up}}$.

We have made all the decisions. Then we recursively solve the two sub-instances and the cost of the instance will be the cost we incurred in the reduction plus the total cost of the two sub-instances. We enumerate all combination of decisions and take the one with the smallest cost.

To convert the above exponential time algorithm to quasi-polynomial time DP, we shall again use the sketching scheme for $\mathcal{J}_{\mathsf{up}}$ defined in Section 3.2. With the sketching scheme, the input to a sub-instance now has small size, since all other parameters can take a few values. Using the same argument as in Section 3.3, we obtain a quasi-polynomial time algorithm for WTar with speed $(8 + \epsilon)$. There is a slight difference between WThr and WTar. In the algorithm for WThr, using the sketching scheme only increases the congestion, but does not affect the approximation ratio. In the algorithm for WTar, the sketching scheme also affect the approximation ratio. It is easy to see that the approximation ratio lost is $1 + \delta$ by using the sketching scheme: in the proof of Lemma 3.5, we constructed a mapping from $\mathcal{J}_1$ to $\mathcal{J}_2$ such that each job in $\mathcal{J}_1$ is mapped to an extent exactly 1, each job in $\mathcal{J}_2$ is mapped to an extent at most $1 + \delta$. Moreover if some $J_1 \in \mathcal{J}_1$ is mapped to some $J_2 \in \mathcal{J}_2$, then J_1 is "easier" than J_2: either $r_{J_1} = r_{J_2}, d_{J_1} \geq d_{J_2}$ or $r_{J_1} \leq r_{J_2}, d_{J_1} = d_{J_2}$. In either case, if we use the same scheduling interval for J_1 and J_2, the tardiness for J_1 will be at most the tardiness for J_2. Use the fact that each job in $\mathcal{J}_2$ is mapped to an extent at most $1 + \delta$, we conclude that the sketching scheme will increase the cost by at most a factor of $1+\delta$. The final approximation ratio we can guarantee is $1 + \epsilon$, even in the case of Tar. Since the proof for our efficient DP is almost identical to the proof for WThr, we omit it here.

We now describe how we handle the case where N and W are super-polynomial. By binary search, we can assume we are given the optimum cost opt. With this opt, we can make two modifications to the input instance, which are already described respectively in the algorithms for WFT and WThr. First, if a job J has $p_J \leq (d_J - r_J)/\mathrm{poly}(n, 1/\epsilon)$, we change the size of p_J to 0. This can only make the problem simpler. Using the similar idea as in Lemma 4.2, we can change the sizes back to the original sizes, with $(1 + \epsilon)$-speed and no loss in the approximation ratio. That is, if the 0-sized job J is scheduled inside (r_J, d_J), we can find an interval for J inside (r_J, d_J). If it is scheduled at $C_J \geq d_J$, then we can find an interval for J inside (r_J, C_J). Second, if $w_J p_J \leq \mathsf{opt}/\mathrm{poly}(n, 1/\epsilon)$, then, we change the size of p_J to 0. By Lemma 5.2, we can change the sizes back by losing a $(1 + \epsilon)$-speed and $(1 + \epsilon)$-approximation.

We define a hard deadline e_J for each job J. The hard deadline e_J is defined as the maximum integer such that $w_J(e_J - d_J) \leq \mathsf{opt}$. That is, if the job J is finished after e_J, then the total cost will be more opt. Notice that if p_J is not 0, then $w_J p_J \geq \mathsf{opt}/\mathrm{poly}(n, 1/\epsilon)$ and $p_J \geq (d_J - r_J)/\mathrm{poly}(n, 1/\epsilon)$. Thus, $(e_J - d_J) \leq \mathsf{opt}/w_J \leq p_J \cdot \mathrm{poly}(n, 1/\epsilon)$, i.e, $p_J \geq (e_J - d_J)/\mathrm{poly}(n, 1/\epsilon)$, implying $p_J \geq (e_J - r_J)/\mathrm{poly}(n, 1/\epsilon)$. We call (r_J, e_J) the hard window for J.

We use the tricks we used in WThr and WFT. When defining the laminar family of permissible intervals, we avoid cutting hard windows of small jobs. We start from the block $(0, N)$ and let it be a permissible interval. We divide $(0, N)$ into two parts and recursively defining permissible intervals in the two parts. Suppose we are now dealing with (A, B), which is a permissible interval. We want to find a point C and define two permissible intervals (A, C) and (C, B). We say a job J is small if its size is positive and at most $(B - A)/\mathrm{poly}(n, 1/\epsilon)$. Then, we choose a C so that C is not inside the hard window of any small job. We can choose a C so that $\min\{C - A, B - C\}$ at least $(1/2 - \epsilon/32)(B - A) - 1/2$ (the -1/2 is for rounding C to a integer). To avoid the dependence of running time on N, we stop recurse if no positive-sized jobs can be completely scheduled in (A, B). By the property that $p_J \geq (e_J - r_J)/\mathrm{poly}(n, 1/\epsilon)$ if $p_J > 0$, it's easy to see that the number of blocks in the laminar family is $\text{poly}(n, 1/\epsilon)$.

We show that for each interval (a, b), we can find a permissible interval in the family with length at least $(1/4 - \epsilon/32)(b - a)$ that are contained in (a, b). To see this, consider the inclusive-minimal block (A, B) in the laminar family that contain (a, b). Suppose (A, B) is divided into (A, C) and (B, C) in the laminar family. Thus $a < C < b$. WLOG we assume $C - a \geq (b - a)/2$. Then (A, C) is recursively divided into two sub-blocks and we always focus on the right sub-block. Consider the first time the block is completely inside (a, C). Assume the block is (A'', C) and its parent block is (A', C). Then, $A' < a \leq A''$. Since A'' split (A', C) almost equally, we have $C - A'' \geq (1/2 - \epsilon/32)(C - A') - 1/2$. Since $C - A' \geq C - a + 1$, we have $C - A'' \geq (1/2 - \epsilon/32)(C - a) - \epsilon/32 \geq (1/2 - \epsilon/16)(C - a) \geq (1/4 - \epsilon/32)(b - a)$. Thus, by using $1/(1/4 - \epsilon/32) \leq (4 + \epsilon)$-speed, we can assume the permissible intervals form a laminar family. Using the same technique as we handle the WThr problem, we can reduce the running time to quasi-polynomial.

7 Lower Bound for Machine Minimization

In this section we show a lower bound of $2^{\log^{1-\epsilon} n}$ on the speed needed to reduce the extra factor of machines to be $o(\log \log n)$ for the discrete variant of the problem for any constant $\epsilon > 0$ unless $\mathtt{NP}$ admits quasi-polynomial time optimal algorithms. To show the lower bound, we extend the result of [12] showing a lower bound of $\Omega(\log \log n)$ on the factor of extra machines needed without speed augmentation. In [12] they create a job scheduling instance where no algorithm can distinguish between whether the instance is feasible on a single machine or $\Omega(\log \log n)$ machines unless $\mathtt{NP} \subseteq \mathtt{DTIME}(n^{\log \log \log n})$. In the case that the instance requires $\Omega(\log \log n)$ machines, they show that in the instance there must be a set of nested time intervals where $\Omega(\log \log n)$ jobs must be scheduled. We build on this result by extending their instance so that not only are there nested time intervals where jobs must be scheduled, but also many such parallel instances in each interval so that even with speed augmentation there must be $\Omega(\log \log n)$ jobs scheduled scheduled at the same time. We do this by adding extra jobs to the instance, but not too many to ensure the instance has size at most $n^{\mathtt{poly}(\log n)}$. We note that the majority of the instance is the same as in [12]. The complete proof of this lower bound result will appear in the full version of this paper.

8 Discussion and Further Applications

In this paper, we developed a novel dynamic programming framework for various non-preemptive scheduling problems. Our framework is very flexible; it applies to WThr, MM, WFT, WTar. We can handle other scheduling problems besides those discussed above. To give a few examples, we sketch algorithms for the following problems.

Maximum Weighted Throughput in Related Machines: The problem is the same as WThr except that machines can have different speeds. A job J can be scheduled in an interval of length p_J/s in (r_J, d_J) on a machine of speed s. We only consider the case when N and W are $\text{poly}(n)$ and speeds are positive integers polynomially bounded by n. By using $(1 + \epsilon)$-extra speed, we may assume the number of different speeds is $O(\log n/\epsilon)$. With this observation, it is straightforward to get a quasi-polynomial time $(1+\epsilon, 1-\epsilon)$-approximation for this problem by modifying our algorithm for WThr.

Convex Cost Function for Flow Time: This problem generalizes many problems where each job's objective grows depending on the job's waiting time. In this problem, each job has a release time but no deadline. Jobs may have weights. We are also given a non-decreasing and non-negative convex function g. Our goal is to minimize $\sum_J w_J g(F_J)$. This general cost function captures WFT and ℓ_k norms of flow time. The problem without requiring g to be convex was studied both in the offline and online settings [4, 20, 14], but all in the preemptive setting. We note that [4] considered an even more general case where each job can have a different cost function. For the case where g is convex, we easily obtain a quasi-polynomial time $(1 + \epsilon, 1 + \epsilon)$-approximation. This is because we can wlog assume jobs of the same type (size and weight) can be scheduled in FIFO order as was the case for WFT. Using the same tricks as in the algorithm for WFT, we can handle the case when N and W are large. It would be interesting to address the problem with a non-convex cost function.

Scheduling with Outliers: In this scenario, we are given a threshold $p \in [0, 1]$ and our goal is to schedule p fraction of jobs in $\mathcal{J}$. Various scheduling objectives can be considered in this scenario, including MM, WFT, WTar, and minimizing some convex cost function for flow time. Optimization problems with outliers were considered in various settings [8, 16], and scheduling problems with outliers were considered in [6, 17]; see [17] for pointers to other scheduling works of similar spirit. In particular, [17] gives a logarithmic approximation for FT in the preemptive setting.

It is fairly straightforward to extend our results to the outlier setting. The only change is to keep the number of type-(i, j) jobs that need to be scheduled in the input for each (i, j) pair. With small modifications to our algorithms for non-outlier problems, we obtain the first set of results for scheduling with outliers in the non-preemptive setting.

We finish with several open problems. Although our framework is flexible enough to give the first or improved non-trivial results for a variety of non-preemptive scheduling problems, it requires quasi-polynomial time. Is it possible to make our algorithm run in polynomial time? Is there a more efficient sketching scheme? Also it would be interesting if one can give a $(1 + \epsilon)$-speed 1-approximation for MM.

Acknowledgements. We thank Kirk Pruhs for pointing us to this problem and having extensive discussions with us. We also thank Alberto Marchetti-Spaccamela, Vincenzo Bonifaci, Lukasz Jez, Andreas Wiese, and Suzanne van der Ster – Kirk brought valuable intuitions which were developed from his discussions with them.

References

[1] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45:501–555, 1998.

[2] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *J. ACM*, 45(1):70–122, 1998.

[3] N. Bansal, Ho-Leung Chan, R. Khandekar, K. Pruhs, B. Schieber, and C. Stein. Non-preemptive min-sum scheduling with resource augmentation. In *Proceedings of the 48th Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 614–624, 2007.

[4] Nikhil Bansal and Kirk Pruhs. The geometry of scheduling. In *FOCS*, pages 407–414, 2010.

[5] Amotz Bar-Noy, Sudipto Guha, Joseph Naor, and Baruch Schieber. Approximating the throughput of multiple machines in real-time scheduling. *SIAM J. Comput.*, 31(2):331–352, 2001.

[6] Moses Charikar and Samir Khuller. A robust maximum completion time measure for scheduling. In *SODA*, pages 324–333, 2006.

[7] Chandra Chekuri and Sanjeev Khanna. Approximation schemes for preemptive weighted flow time. In *STOC*, pages 297–305, 2002.

[8] Ke Chen. A constant factor approximation algorithm for k-median clustering with outliers. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 826–835, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[9] Julia Chuzhoy and Paolo Codenotti. Resource minimization job scheduling. In *APPROX-RANDOM*, pages 70–83, 2009.

[10] Julia Chuzhoy and Paolo Codenotti. Resource minimization job scheduling [erratum], 2013.

[11] Julia Chuzhoy, Sudipto Guha, Sanjeev Khanna, and Joseph (Seffi) Naor. Machine minimization for scheduling jobs with interval constraints. In *Proceedings of the 45th Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 81–90, 2004.

[12] Julia Chuzhoy and Joseph (Seffi) Naor. New hardness results for congestion minimization and machine scheduling. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 28–34, 2004.

[13] Julia Chuzhoy, Rafail Ostrovsky, and Yuval Rabani. Approximation algorithms for the job interval selection problem and related scheduling problems. *Math. Oper. Res.*, 31(4):730–738, 2006.

[14] Kyle Fox, Sungjin Im, Janardhan Kulkarni, and Benjamin Moseley. Online non-clairvoyant scheduling to simultaneously minimize all convex functions. In *APPROX-RANDOM*, pages 142–157, 2013.

[15] M. R. Garey and D. S. Johnson. Two processor scheduling with start times and deadline. *SIAM Journal on Computing*, 6:416–426, 1977.

[16] Naveen Garg. Saving an epsilon: a 2-approximation for the k-mst problem in graphs. In *STOC*, pages 396–402, 2005.

[17] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Danny Segev. Scheduling with outliers. In *APPROX-RANDOM*, pages 149–162, 2009.

[18] Wiebke Höhn, Julián Mestre, and Andreas Wiese. How unsplittable-flow-covering helps scheduling with job-dependent cost functions. In *ICALP (1)*, pages 625–636, 2014.

[19] Sungjin Im, Benjamin Moseley, and Kirk Pruhs. A tutorial on amortized local competitiveness in online scheduling. *SIGACT News*, 42(2):83–97, 2011.

[20] Sungjin Im, Benjamin Moseley, and Kirk Pruhs. Online scheduling with general cost functions. In *SODA*, pages 1254–1265, 2012.

[21] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47:617–643, 2000.

[22] Hans Kellerer, Thomas Tautenhahn, and Gerhard J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *In Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 418–426, 1995.

[23] Kirk Pruhs, Jiri Sgall, and Eric Torng. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter Online Scheduling. 2004.

[24] P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Cobminatorica*, 7:365–374, 1987.