

# Look Before, Before You Leap: Online Vector Load Balancing with Few Reassignments

Varun Gupta ✉

Booth School of Business, University of Chicago, IL, USA

Ravishankar Krishnaswamy ✉

Microsoft Research, Bengaluru, India

Sai Sandeep ✉

University of California, Berkeley, CA, USA

Janani Sundaresan ✉

Department of Computer Science, Rutgers University, Piscataway, NJ, USA

---

## Abstract

---

In this paper we study two *fully-dynamic* multi-dimensional vector load balancing problems *with recourse*. The adversary presents a stream of  $n$  job insertions and deletions, where each job  $j$  is a vector in  $\mathbb{R}_{\geq 0}^d$ . In the *vector scheduling* problem, the algorithm must maintain an assignment of the active jobs to  $m$  identical machines to minimize the makespan (maximum load on any dimension on any machine). In the *vector bin packing* problem, the algorithm must maintain an assignment of active jobs into a number of bins of unit capacity in all dimensions, to minimize the number of bins currently used. In both problems, the goal is to maintain solutions that are competitive against the optimal solution for the active set of jobs, at every time instant. The algorithm is allowed to change the assignment from time to time, with the secondary objective of minimizing the amortized recourse, which is the average cardinality of the change of the assignment per update to the instance.

For the vector scheduling problem, we present two simple algorithms. The first is a randomized algorithm with an  $O(1)$  amortized recourse and an  $O(\log d / \log \log d)$  competitive ratio against oblivious adversaries. The second algorithm is a deterministic algorithm that is competitive against adaptive adversaries but with a slightly higher competitive ratio of  $O(\log d)$  and a per-job recourse guarantee bounded by  $\tilde{O}(\log n + \log d \log \text{OPT})$ . We also prove a sharper instance-dependent recourse guarantee for the deterministic algorithm.

For the vector bin packing problem, we make the so-called *small jobs* assumption that the size of all jobs in all the coordinates is  $O(1/\log d)$  and present a simple  $O(1)$ -competitive algorithm with  $O(\log n)$  recourse against oblivious adversaries.

For both problems, the main challenge is to determine when and how to migrate jobs to maintain competitive solutions. Our central idea is that for each job, we make these decisions based only on the active set of jobs that are “earlier” than this job in some ordering  $\prec$  of the jobs.

**2012 ACM Subject Classification** Theory of computation → Scheduling algorithms

**Keywords and phrases** Vector Scheduling, Vector Load Balancing

**Digital Object Identifier** 10.4230/LIPIcs.ITCS.2023.65

**Funding** *Sai Sandeep*: Work supported in part by NSF grant CCF-2228287.

*Janani Sundaresan*: Work partly supported by Microsoft Research India.

**Acknowledgements** We want to thank the anonymous reviewers at ITCS for their helpful comments and suggestions.

## 1 Introduction

In this paper, we consider two central problems in dynamic allocation of multi-dimensional jobs to machines. As a motivation, consider a common scenario in data center scheduling: there are a set of identical machines each with a specified capacity vector over  $d$  different



© Varun Gupta, Ravishankar Krishnaswamy, Sai Sandeep, and Janani Sundaresan; licensed under Creative Commons License CC-BY 4.0

14th Innovations in Theoretical Computer Science Conference (ITCS 2023).

Editor: Yael Tauman Kalai; Article No. 65; pp. 65:1–65:17



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

resources such as CPU, DRAM, SSD, network bandwidth, etc. A set of jobs arrive to and depart from the system, with each job being characterized by its resource requirement in each of the  $d$  dimensions. The goal is to dynamically maintain an allocation of jobs to machines in a load-balanced manner while making as few reassignments as possible. There are two classical notions of formalizing the goal of balanced assignment: *vector scheduling* and *vector bin packing*.

► **Problem 1** (ONLINEVECTORSCHEDED). *An instance to the problem consists of a collection  $M$  of  $m$  identical machines with  $d$  resources, and a set of active jobs  $A(t)$  at each time  $t$ , with  $A(0) = \emptyset$ . At time  $t = 1, 2, \dots, n$ ,  $A(t)$  is updated by either the insertion of a new job  $j$ , or the removal of an existing job in  $A(t-1)$ . Each arriving job  $j$  is characterized by a load vector  $\mathbf{p}(j) \in \mathbf{R}_{\geq 0}^d$  indicating its resource requirements. The goal is to maintain an assignment  $f_t : A(t) \rightarrow M$  to minimize the makespan at time  $t$ ,  $\max_{m \in M} \max_{k \in [d]} \sum_{j \in A(t): f(j)=m} p_k(j)$ . The algorithm can reassign jobs from one machine to another, and the total number of times any job is reassigned is called its recourse.*

► **Problem 2** (ONLINEVECTORBINPACK). *An instance to the problem consists of a collection  $M$  of identical machines/bins with  $d$  resources each and a capacity of 1 for each resource, and a set of active jobs  $A(t)$  at each time  $t$ , with  $A(0) = \emptyset$ . At each time  $t = 1, 2, \dots, n$ ,  $A(t)$  is updated by either the insertion of a new job  $j$ , or the removal of an existing job in  $A(t-1)$ . Each arriving job  $j$  is characterized by a load vector  $\mathbf{p}(j) \in \mathbf{R}_{\geq 0}^d$  indicating its different resource requirements. The goal is to maintain an assignment  $f_t : A(t) \rightarrow M$  to minimize the number  $|f_t(A(t))|$  of machines used,<sup>1</sup> while ensuring that each machine is feasible, i.e.,  $\max_{m \in M} \max_{k \in [d]} \sum_{j \in A(t): f(j)=m} p_k(j) \leq 1$ . The algorithm can reassign jobs from one machine to another, and the total number of times any job is reassigned is called its recourse.*

► **Definition 3.** *We say that an online algorithm is  $(\alpha, \beta)$ -competitive for ONLINEVECTORSCHEDED if for all times  $t$ , the cost of the algorithm's schedule  $f_t$  is at most  $\alpha$  times the cost of the optimal makespan for  $A(t)$ , and the total recourse over all jobs is at most  $\beta t$ . Similarly, an online algorithm is  $(\alpha, \beta)$ -competitive for ONLINEVECTORBINPACK if for all times  $t$ , the number of bins used by the algorithm to pack  $A(t)$  is at most  $\alpha$  times that of the optimal packing for the jobs  $A(t)$ , and the total recourse over all jobs is at most  $\beta t$ .*

One motivation for competitive vector scheduling against the current optimal comes from bandwidth sharing [20]. Assume that the machines correspond to  $m$  network connections, and the jobs correspond to video streaming requests. Since the network connections have limited bandwidth, a common technique to handle multiple streams is to throttle the video quality, where the quality of all the jobs sharing a network connection is a decreasing function of the total load of the network connection. It is now quite reasonable to see that competitive guarantees against peak optimal are unsatisfactory. A motivation for dynamic bin packing comes from real-time capacity provisioning of data centers where the goal is to aggregate the active workload on a few servers and power off idle servers to minimize energy cost. In both settings, migration of jobs causes overheads, and therefore minimizing the recourse is an important metric. However, beyond these two examples, vector scheduling and packing are fundamental problems that appear as a motif in many applications and are central objects of study in theoretical computer science where they have been studied both in offline and online settings and in scalar and vector variants. Our work is the first that fills an important gap in this literature – on vector scheduling and packing with both arrivals and departures.

<sup>1</sup> Here we use the notation that  $f(X)$  is the range  $|\cup_{j \in A(t)} f(j)|$  of the function over the input  $X$ .

## 1.1 Results and Paper Outline

In Section 2 we consider the fully dynamic online vector scheduling problem, and present a randomized algorithm with  $O\left(\frac{\log d}{\log \log d}\right)$  competitive ratio and  $O(1)$  amortized recourse against oblivious adversaries.

► **Theorem 4.** *There is an efficient randomized  $(O(\log d / \log \log d), O(1))$ -competitive algorithm for ONLINEVECTORSCHEd against oblivious adversaries (where the sequence of job insertions and deletions is independent of the algorithm’s random choices).*

We remark that even in the only insertions setting, there is an optimal lower bound of  $\Omega\left(\frac{\log d}{\log \log d}\right)$  [16] on the competitive ratio of online algorithms for Vector Scheduling (with no recourse). Thus, by using a small  $O(1)$  amortized recourse, our algorithm matches the same  $O\left(\frac{\log d}{\log \log d}\right)$  factor competitive ratio while being able to handle deletions as well.

Often, however, it is a desirable characteristic to have algorithms competitive against adaptive adversaries, since the clients/jobs could selfishly try to tailor their job’s load vector by splitting or aggregating depending on the state of the system. To address this concern, we obtain a simple deterministic algorithm that is competitive against adaptive adversaries, albeit with a slightly worse recourse. We present this algorithm in Section 4.

► **Theorem 5.** *There is an efficient deterministic  $O(\log d)$ -competitive algorithm for ONLINEVECTORSCHEd against adaptive adversaries where any job  $j$  has recourse at most  $\tilde{O}(\log n_{max} + \log d \log \frac{\text{Opt}_{max}}{\text{Opt}_{min}})$ . The quantities  $\text{Opt}_{max}$  and  $\text{Opt}_{min}$  are the maximum and minimum values, respectively, of the optimal makespan during the lifespan of  $j$ , and  $n_{max}$  denotes the maximum size of the active set of jobs during the lifespan of  $j$ .*

In Section 3, we study the fully dynamic online vector bin packing problem. For the ONLINEVECTORBINPACK problem with general vectors, even the offline problem has strong lower bounds of  $\Omega(d^{1-\epsilon})$  on the approximation ratio in a complexity-theoretic sense due to connections with graph coloring [8, 9], and in the online setting, an information-theoretic lower bound of  $\Omega(d^{1-\epsilon})$  was shown by [4]. Hence, the prior works [3] for ONLINEVECTORBINPACK designed online algorithms with  $e + O(\epsilon)$ -competitive ratio for the ONLINEVECTORBINPACK problem with the *small-jobs assumption*, where every  $0 \leq p_k(j) \leq \Theta(\frac{1}{\log d})$ , when there are arrivals only. We extend this framework to design fully dynamic algorithms with  $O(1)$ -competitive ratio and  $O(\log n)$  recourse for the ONLINEVECTORBINPACK problem with the small-jobs assumption.

► **Theorem 6.** *There is an efficient randomized  $(O(1), O(\log n))$ -competitive algorithm for ONLINEVECTORBINPACK against oblivious adversaries when all the jobs are small.*

### Key Algorithmic Technique

All our algorithms build on the state-of-the-art online algorithms for the arrival-only versions of the vector scheduling and vector bin packing problem, and the competitive ratios of all our algorithms match the known competitive ratios for the arrival-only versions. However, a naive implementation of the existing algorithms would only give guarantees against the peak value of the optimal, which is defined by  $\widehat{\text{Opt}}(t) := \max_{t' \leq t} \text{Opt}(t')$ , since they all rely on some form of guessing-and-doubling ideas baked into the algorithms which need monotonicity of the optimal value. Moreover, a straightforward generalization of the existing online arrival-only algorithms to make them competitive against the current optimal  $\text{Opt}(t)$  of the active set of jobs potentially leads to a large recourse.

To overcome these hurdles, our crucial idea is to maintain a total ordering  $\prec_t$  of all the jobs currently in the system at time  $t$ , and whenever we need to make a recourse decision for a job  $j$ , we only look at the jobs which are “earlier” than job  $j$  in this ordering. As an example, in the randomized algorithm for vector scheduling, the ordering  $\prec_t$  is simply given by the arrival order of jobs. To make the recourse decision for a job  $j$  we compare the load on its machine due to jobs  $j' \prec_t j$  and an estimate of  $\text{Opt}(t)$  restricted to this set of jobs. The monotonic non-increasing property of these two quantities during the lifetime of  $j$  allows for a low recourse and a much more tractable analysis. We believe this could serve as a generic recipe to convert arrivals-only scheduling algorithms to fully dynamic scheduling algorithms with bounded recourse, with the choice of the ordering  $\prec_t$  being problem-dependent.

## 1.2 Related Work

In the offline setting, for the Vector Scheduling problem, Chekuri and Khanna [8] gave a  $O(\log^2 d)$  factor approximation algorithm. This has been improved to  $O\left(\frac{\log d}{\log \log d}\right)$  by Harris, Srinivasan [15], and Im, Kell, Kulkarni, Panigrahi [16]. Very recently, almost matching hardness of  $\Omega((\log d)^{1-\epsilon})$  has been obtained by Sandeep [19]. For the Vector Bin Packing problem, the asymptotic PTAS for 1-Dimensional Bin Packing [12] can be extended to give a  $d$ -factor approximation algorithm, and almost matching hardness has been obtained by Chekuri, Khanna [8, 9]. When  $d$  is a fixed constant, improved algorithms with approximation ratio  $\log d + O(1)$  have been obtained [8, 5, 6] and matching hardness (up to constants) has been proved recently [19].

Online algorithms for Vector Scheduling and Vector Bin Packing with only insertions have been extensively studied in the literature. For the Vector Scheduling problem, Meyerson, Roytman and Tagiku [17] gave a  $O(\log d)$  competitive online algorithm. This has been improved to  $O\left(\frac{\log d}{\log \log d}\right)$  by Im, Kell, Kulkarni, and Panigrahi [16] who also showed a matching lower bound. For the Vector Bin Packing problem, the simple first-fit algorithm [14] gives a competitive ratio of  $O(d)$  and an almost matching lower bound of  $\Omega(d^{1-\epsilon})$  was given by Azar, Cohen, Kamara, and Shepherd [4]. Algorithms for Vector Scheduling and Bin Packing have also been studied in other settings such as streaming algorithms [11]. The study of the fully dynamic model for bin packing was initiated in [10] where the metric of comparison was the peak optimal. Most recently, [13] proposed a constant competitive algorithm against the current optimal for the scalar version of fully dynamic bin packing. For fully dynamic scheduling in the scalar case, Westbrook [20] was the first to give competitive algorithms with low approximation ratio against the current optimal for both identical and related machines.

## 2 Randomized algorithm for Vector Scheduling

In this section, we present a simple randomized algorithm for fully dynamic vector scheduling against an adversary who does not have access to the randomness of the algorithm (an oblivious adversary). At a high level, our algorithm builds on that of [16] for online vector scheduling with only arrival of jobs. We first briefly discuss the algorithm in [16], which we call **Random+Greedy**. The algorithm assumes that the maximum load of an optimal solution is at most 1 (by using a standard guess-and-double trick to guess the value of  $\text{Opt}$  and normalizing all coordinates by  $\text{Opt}$ ).

The algorithm maintains *two copies* of the  $m$  machines  $M_R$  and  $M_S$ , and places every incoming job into one of the two copies. Every arriving job chooses a uniformly random machine  $r(j)$  in  $M_R$  as its tentative choice. If the total load of all jobs scheduled on  $r(j)$  is

at most a *fixed threshold* of  $\lambda$  in every dimension, then the algorithm freezes this assignment. On the other hand, if some dimension's load exceeds  $\lambda$ , then the job is instead inserted to  $M_S$ , where it is scheduled in a greedy manner. By definition, all the machines in  $M_R$  have load at most  $\lambda$  in every coordinate, and moreover, a simple Chernoff bound shows that, because  $\text{Opt} \leq 1$ , the probability of any job being inserted in  $M_S$  is at most  $\Theta(1/d)$  for  $\lambda = \Theta(\log d / \log \log d)$ . Therefore, we get that the total load on any machine in  $M_S$  in any coordinate is at most  $d \cdot \Theta(1/d) \leq \Theta(1)$  since the greedy algorithm is  $O(d)$ -competitive for  $d$ -dimensional vector scheduling. By simply combining the copies  $M_R$  and  $M_S$ , we get a schedule with competitive ratio of  $(\lambda + O(1)) = \Theta(\log d / \log \log d)$ .

## 2.1 Hurdles and Resolutions

Our main contribution in this section is what we think is the right generalization of this algorithm for the fully-dynamic vector scheduling problem when jobs can arrive and depart. We first discuss the issues we encounter. Indeed, firstly, we can no longer employ the guess-and-double strategy (and other normalization ideas based on  $\text{Opt}$ ) which the prior algorithms for insertion do, since the  $\text{Opt}(t)$  is a non-monotone quantity when there are deletions at play. Secondly, a priori there is no clear choice of the threshold  $\lambda$  above, and also which jobs to migrate if a machine in  $M_R$  incurs a large load due to  $\text{Opt}(t)$  dropping a lot.

We resolve the first issue by not performing any normalizations and working with the current estimates of  $\text{Opt}(t)$  at all times  $t$ . To deal with the second issue (of lacking a clear threshold  $\lambda$ ), we instead use a job-dependent threshold  $\lambda_j \approx \lambda \cdot \text{Opt}(\preceq j)$  where  $\text{Opt}(\preceq j)$  is the optimal makespan when considering *only the active jobs which arrived before  $j$* . This helps us maintain low-recourse solutions as jobs arrive and depart since this quantity is non-increasing during the entire period in which  $j$  is active. Indeed, in our algorithm, a job stays in  $M_R$  for some amount of its lifetime, and then permanently switches over to the greedy copy  $M_S$  when the load criterion on its random machine exceeds the dynamic value of  $\lambda_j$ .

While this is sufficient for recourse, we also need to ensure that the probability that a job ever switches to  $M_S$  is at most  $O(1/d)$ , since the greedy algorithm is only  $O(d)$ -competitive. Naively using a threshold of  $L = \Theta(\log(dT))$ , where  $T$  is the number of time steps during which a job is active, will allow us to use a union bound over all these time steps but will result in a competitive ratio of  $\Theta(\log(dT))$ . To avoid the dependence on  $T$ , our idea is to check if a job needs to migrate *only at certain points of time* to get a geometric series which converges to  $\Theta(1/d)$  for  $L = \Theta\left(\frac{\log d}{\log \log d}\right)$ . This requires us to control the variance of the desired random variables, for which we employ a standard idea of partitioning jobs into classes based on the value of the maximum size in any dimension. Indeed, there are examples where without this additional step, the competitive ratio would worsen to  $\Omega(d)$  against the current optimal cost.

## 2.2 Algorithm Description

We maintain a total ordering  $\prec_t$  of the active jobs  $A(t)$  at any time step, given by the arrival times of the jobs. The crucial idea in our algorithm to ensure bounded recourse is that every job  $j$  makes its high-level decisions purely based on the set of jobs  $\{j' : j' \preceq_t j\}$ , which forms a monotonically decreasing set during  $j$ 's lifespan. We use a parameter  $\lambda = 3C \frac{\log d}{\log \log d}$  for a suitable constant  $C$ .

Our algorithm partitions the active jobs  $A(t)$  into two kinds: regular jobs  $R(t)$  and spillover jobs  $S(t)$ . We maintain separate scheduling assignments for  $R(t)$  and for  $S(t)$  using different algorithms. For convenience, we keep two copies of the  $m$  machines, with the first copy denoted as  $M_R$  (for random assignments), and the second copy denoted as  $M_S$  (for greedy assignments). We drop the  $t$  subscript when the context is clear.

**Step 1: Rounding and Partitioning into Classes.** For a cleaner presentation, we round up the load values  $p_k(j)$  for every  $j$  and  $k \in [d]$  to the nearest power of 2. This operation incurs at most a 2 factor increase in the competitive ratio. We partition the jobs based on  $\max_{k \in [d]} p_k(j)$ , i.e., we place a job  $j$  in class  $l$ , i.e.,  $j \in \mathcal{C}_l$  if and only if  $\max_{k \in [d]} p_k(j) = 2^l$ . For any job  $j$ , we let  $\text{cl}(j)$  denote its class. Note that for all jobs  $j \in \mathcal{C}_l$  and all dimensions  $k$ ,  $p_k(j) \leq 2^l$ .

**Defining Job-Dependent Migration Thresholds.** We formally define some thresholds for every job, using which we shall decide whether to place a job in  $R(t)$  or migrate it to  $S(t)$ . To this end, we consider the following dynamic thresholds. In our algorithm, every job will be assigned a random value  $\sigma(j) \in [m]$  (essentially a random machine choice), which does not change over time. For every job  $j \in A(t)$  and dimension  $k \in [d]$ , we define the following parameters.

$$\text{ExpLoad}_k(j) = \frac{\sum_{j' \prec_t j, \text{cl}(j) = \text{cl}(j')} p_k(j')}{m}$$

$$\text{ActLoad}_k(j) = \sum_{j' \prec_t j, \text{cl}(j) = \text{cl}(j'), \sigma(j) = \sigma(j')} p_k(j')$$

The idea is that since  $\sigma$  is a random choice,  $\text{ActLoad}_k(j)$  should be comparable to  $\text{ExpLoad}_k(j)$ , and can exceed  $\lambda \text{ExpLoad}_k(j)$  with low probability.

**Step 2: Algorithm for Insertion and Deletion of Jobs.** We now present our full algorithm for insertions and deletion of jobs in Algorithm 1 and Algorithm 2.

■ **Algorithm 1** Insert(job  $j$  with load vector  $\mathbf{p}(j) \in \mathbf{R}_{\geq 0}^d$ ).

- 
- 1: Choose a machine  $\sigma(j)$  uniformly at random from  $[m]$ .
  - 2: Compute  $\text{cl}(j)$  and the values  $\text{ExpLoad}_k(j)$  and  $\text{ActLoad}_k(j)$  w.r.t  $A(t)$  and  $\preceq_t$ .
  - 3: Set job thresholds to be  $\tau_k(j) = (\text{ExpLoad}_k(j) + 2^{\text{cl}(j)})$  for every dimension  $k$ .
  - 4: **if**  $\text{ActLoad}_k(j) \leq \lambda \cdot \tau_k(j)$  for every  $k \in [d]$  **then**
  - 5:     Assign  $j$  to the machine  $\sigma(j)$  and place  $j \in R(t)$ .
  - 6: **else**
  - 7:     Insert  $j$  to the Spillover instance  $S(t)$ .
  - 8: **end if**
- 

**Spillover section algorithm.** For the spillover instance, whenever our algorithm inserts a job  $j$  into  $S(t)$ , we simply flatten the vector into a 1-dimensional load  $\bar{p}_j = 2^{\text{cl}(j)}$ , and feed it as an insertion into the fully-dynamic 1-dimensional load balancing algorithm of Westbrook [20, Section 2]. Similarly, when our algorithm deletes a job  $j$  from  $S(t)$ , we simply delete the 1-dimensional job  $\bar{p}_j$  from the spillover instance using the same algorithm. Internally, upon every insertion or deletion, this algorithm might perform some recourse, but the overall amortized recourse is at most  $O(1)$  per update. Moreover, this algorithm is  $O(1)$ -competitive w.r.t the current optimal of the spillover instance.



---

**Algorithm 2** Delete(job  $j$ ).
 

---

```

1: if  $j$  is in the Spillover instance  $S(t)$  then
2:   Delete  $j$  from the Spillover instance which is running the 1d-Alg.
3: else
4:   for all active jobs  $j' \succ_t j$  which are in  $R(t)$  do
5:     if for all  $k$ , current threshold  $\text{ExpLoad}_k(j') + 2^{\text{cl}(j')} \geq \tau_k(j')/2$  then
6:       Do nothing.
7:     else
8:       Update job thresholds  $\tau_k(j') = (\text{ExpLoad}_k(j') + 2^{\text{cl}(j')})$  for every dimension  $k$ .
9:       if  $\text{ActLoad}_k(j') > \lambda \cdot \tau_k(j')$  for some  $k \in [d]$  then
10:        Remove  $j'$  from  $R(t)$  and add it to  $S(t)$ 
11:        Remove  $j'$  from current machine  $\sigma(j')$ , and insert it to the Spillover instance.
12:      end if
13:    end if
14:  end for
15: end if

```

---

## 2.3 Analysis

### 2.3.1 Competitive ratio analysis

The first step is to bound the probability that a job ever becomes a spillover job and gets assigned to  $S(t)$ . Towards that end, we begin by recalling the following Chernoff-Hoeffding bound.

► **Theorem 7** (See e.g., [18]). *Let  $X_1, X_2, \dots, X_n$  be independent Bernoulli random variables and let  $a_1, a_2, \dots, a_n$  be coefficients in  $[0, 1]$ . Let  $X = \sum_i a_i X_i$ , and let  $\mu = \mathbb{E}[X]$ . Then, for every  $\delta > 0$ ,*

$$\Pr[X > (1 + \delta)\mu] \leq \left( \frac{e^\delta}{(1 + \delta)^{1 + \delta}} \right)^\mu$$

Using Theorem 7, we obtain the following concentration inequality regarding the linear summation of Bernoulli random variables.

► **Lemma 8.** *Fix integers  $d, n, m \geq 2$ . Suppose that  $X_1, X_2, \dots, X_n$  are independent Bernoulli random variables each taking value 1 with probability equal to  $\frac{1}{m}$ . Let  $Y := a_1 X_1 + a_2 X_2 + \dots + a_n X_n$  be a random variable where  $a_i$ s are positive reals. Let  $L_1 := \frac{\sum_{i=1}^n a_i}{m}$  and  $L_2 := \max_{i \in [n]} a_i$ , and  $\lambda = 3C \frac{\log d}{\log \log d}$  for an absolute constant  $C$ . Then,*

$$\Pr(Y \geq \lambda(L_1 + L_2)) \leq \left( \frac{1}{d} \right)^{\frac{C(L_1 + L_2)}{L_2}}$$

We now prove a key lemma that we use to bound the competitive ratio of the algorithm.

► **Lemma 9.** *At any point of time, for any job  $j$ , the probability that  $j$  is in the Spillover section is at most  $\frac{1}{d}$ .*

**Proof.** Note that the job  $j$  is moved to the Spillover section if at some point in the execution of the algorithm, for some  $k$ , we have  $\text{ActLoad}_k(j) > \lambda \cdot \tau_k(j) = \lambda(\text{ExpLoad}_k(j) + 2^{\text{cl}(j)})$ , either in Algorithm 1 in Algorithm 1, or when some job  $j \prec j'$  gets deleted in Algorithm 2 in Algorithm 2.

By using Lemma 8 on the set of jobs  $\{j' \preceq_t j : \text{cl}(j') = \text{cl}(j)\}$ , we get that the probability of this happening is at most  $\left(\frac{1}{d}\right)^C \frac{\text{ExpLoad}_k(j) + 2^{\text{cl}(j)}}{2^{\text{cl}(j)}}$ . Here, we are using the fact that the assignments of machines to the jobs are mutually independent, and since we are operating in the oblivious adversary setting, the machine assignments remain independent throughout the execution of the algorithm. Moreover, this is the place where we use the fact that all jobs in class  $\text{cl}(j)$  have all load in any dimension bounded by  $2^{\text{cl}(j)}$ , and this is why the algorithm only considers jobs of the same class for migration.

Note that the above probability is at most  $\left(\frac{1}{d}\right)^C$  for every value of  $\text{ExpLoad}_k(j)$  and  $2^{\text{cl}(j)}$ . Furthermore, suppose that  $\frac{\text{ExpLoad}_k(j)}{2^{\text{cl}(j)}} = \alpha$ , then the above probability is equal to  $\left(\frac{1}{d}\right)^{C(1+\alpha)}$ . Finally, observe that for any dimension, this check occurs only when the threshold  $\tau_k(j)$  halves, hence we get that the  $1 + \alpha$  value has to be halving, eventually ending up at 1. Thus, for every  $k$ , the probability that the event  $\text{ActLoad}_k(j) > \lambda(\text{ExpLoad}_k(j) + 2^{\text{cl}(j)})$  happens at any check point in the execution of the algorithm is at most

$$\left(\frac{1}{d}\right)^C + \left(\frac{1}{d}\right)^{2C} + \left(\frac{1}{d}\right)^{4C} + \dots \leq \left(\frac{1}{d}\right)^{C-1}$$

Taking a further union bound over all dimensions  $k \in [d]$ , we get the required claim when we set  $C$  to be large enough.  $\blacktriangleleft$

We are now ready to bound the competitive ratio of the algorithm.

► **Lemma 10.** *At any point in time, the expected maximum load of the algorithm is at most  $\lambda + O(1)$  times the current optimal cost.*

**Proof.** Fix a time  $t$ , machine  $i \in [m]$  and dimension  $k \in [d]$ . We bound the load on the Random segment and the Spillover segment separately. Note that if a job  $j$  is in the Random segment, we have

$$\text{ActLoad}_k(j) \leq 2\lambda(\text{ExpLoad}_k(j) + 2^{\text{cl}(j)})$$

for every  $k \in [d]$ , because the condition  $\text{ActLoad}_k(j) \leq \lambda(\text{ExpLoad}_k(j) + 2^{\text{cl}(j)})$  is satisfied if the job chose to remain in  $R(t)$  at the last time the check happened either in Algorithm 1 in Algorithm 1, or when some job  $j \prec j'$  gets deleted in Algorithm 2 in Algorithm 2. Moreover, subsequently, the threshold has not halved (else we would have run Algorithm 2 in Algorithm 2 with the updated threshold). For an integer  $l$ , we let  $j_l^{\max}$  be the latest job that has arrived that is in the partition  $l$ :  $j_l^{\max} := \max\{j \in R(t) : j \in \mathcal{C}_l\}$ .

We first bound the load (henceforth referred to as  $\text{load}_r$ ) on machine  $i \in [m]$  at time  $t$  in dimension  $k$  in the Random segment.

$$\begin{aligned} \text{load}_r &= \sum_{j \in R(t) : \sigma(j) = i} p_k(j) \\ &\leq \sum_{l : \mathcal{C}_l \neq \emptyset} (\text{ActLoad}_k(j_l^{\max}) + 2^l) \\ &\leq \sum_{l : \mathcal{C}_l \neq \emptyset} (2\lambda(\text{ExpLoad}_k(j_l^{\max}) + 2^l) + 2^l) \\ &\leq 2\lambda \sum_l \frac{\sum_{j \in \mathcal{C}_l} p_k(j)}{m} + 3\lambda \sum_{l : \mathcal{C}_l \neq \emptyset} 2^l \\ &\leq 2\lambda \frac{\sum_{j \in R(t)} p_k(j)}{m} + 6\lambda \max_{j \in R(t)} \max_{k \in [d]} p_k(j) \\ &\leq O(\lambda) \cdot \text{Opt}(t). \end{aligned}$$



In the first inequality above, we bound the total load in each class in each dimension by the active load of the last job (plus its own load). In the second inequality, we use the stability condition of why a job stays in the set  $R(t)$ . The third inequality uses the definition of  $\text{ExpLoad}$ , and the last one uses the straightforward lower bound on  $\text{Opt}(t) \geq \max_k \frac{\sum_{j \in A(t)} p_k(j)}{m}$  and  $\text{Opt}(t) \geq \max_k \max_{j \in A(t)} p_k(j)$ .

We now bound the load (henceforth referred to as  $\text{load}_s$ ) on machine  $i \in [m]$  at time  $t$  in dimension  $k$  in the Spillover segment. Note that by the algorithm [20, Section 2], we get that the maximum load on any machine in the Spillover segment is at most  $O(1)$  times the sum of the average load and maximum load of a machine. Thus, we get

$$\text{load}_s \leq O(1) \left( \sum_{j \in S(t)} \frac{2^{\text{cl}(j)}}{m} + \max_{j \in S(t)} 2^{\text{cl}(j)} \right) \leq O(1) \left( \sum_{j \in S(t)} \frac{2^{\text{cl}(j)}}{m} + \text{Opt}(t) \right)$$

Using Lemma 9, we get

$$\begin{aligned} \mathbb{E}[\text{load}_s] &\leq O(1) \left( \sum_{j \in A(t)} \frac{2^{\text{cl}(j)}}{dm} + \text{Opt}(t) \right) \\ &\leq O(1) \left( \sum_{j \in A(t)} \frac{\sum_{k \in [d]} p_k(j)}{dm} + \text{Opt}(t) \right) \\ &\leq O(1) \left( \frac{\max_{k \in [d]} \sum_{j \in A(t)} p_k(j)}{m} + \text{Opt}(t) \right) \\ &\leq O(1) \text{Opt}(t) \end{aligned}$$

Thus, we get that the expected load on any machine in any dimension is at most  $O(\lambda)$  times the current optimal maximum load  $\text{Opt}(t)$ . ◀

### 2.3.2 Recourse analysis

Finally, we analyze the recourse of the algorithm.

► **Lemma 11.** *The amortized recourse of this algorithm is  $O(1)$  over a sequence of arrivals and departures. Moreover, reassignments happen only when some job  $j$  departs.*

**Proof.** During any insertion, the incoming job is assigned to either  $R(t)$  or  $S(t)$  and no job is reassigned to any other machine. If the job  $j$  stays in the Random segment, it does not change its assignment throughout the course of execution of the algorithm. At some point, it might move to the Spillover segment, where it is treated as an insertion to the 1-dimensional greedy load balancing algorithm of [20, Section 2]. From [20, Section 2], the amortized number of reassignments for this greedy algorithm is at most 2. Thus, overall, in expectation, the amortized number of reassignments of our overall algorithm is at most  $O(1)$  per update. ◀

### 3 Fully Dynamic Vector Bin Packing

#### 3.1 Algorithm Description

We maintain a total ordering  $\prec_t$  of the active jobs  $A(t)$  given by the arrival times of the jobs. As in Section 2, the crucial idea is to ensure any job makes its migration decisions purely based on the set of jobs  $\{j' : \preceq_t j\}$ . Since this criterion is not affected by jobs that arrive subsequently, and only changes monotonically (when jobs arriving earlier are deleted), the recourse of our algorithm can be bounded favorably.

In more detail, our algorithm partitions the active jobs  $A(t)$  into two kinds: regular jobs  $R(t)$  and spillover jobs  $S(t)$ . We maintain separate bin packing assignments for  $R(t)$  and for  $S(t)$  using different algorithms. For convenience, we keep a distinct set of indices (to represent the ID of the bin) for the two kinds – we use the bins numbered  $1, 2, \dots$  for the regular jobs, and a different range of indices cover the spillover bins. At times, jobs might migrate between the two kinds, but we shall bound the total number of migrations. Our assignment within  $R(t)$  will not perform any recourse, while our assignment algorithm within  $S(t)$  will perform  $O(1)$ -amortized recourse per update to  $S(t)$ .

With a slight abuse of notation, let  $\text{load}_t^k(\preceq_t j) = \sum_{j' \preceq_t j} p_k(j')$  denote the total load on dimension  $k$  for jobs occurring before (and including)  $j$  in the ordering  $\prec_t$ . We divide the bins serving the regular jobs into *partitions*, with partition  $\mathcal{P}_i$  containing  $2^{i-1}$  bins indexed  $\{2^{i-1}, 2^{i-1} + 1, \dots, 2^i - 1\}$  for  $i \geq 1$ . A crucial component of our algorithm is the following procedure, **SetPart**, which determines which partition a job is placed into at all points in time.

■ **Algorithm 3** **SetPart**(job  $j$  with load vector  $\mathbf{p}(j) \in \mathbf{R}_{\geq 0}^d$ ).

- 
- 1: Set  $\text{part}(j)$  to be  $\text{argmin } i \geq 1$  s.t.  $\max_k \text{load}_t^k(\preceq_t j) \leq 2^{i-2}$ , i.e., smallest  $i \geq 1$  such that the total load on every dimension of jobs up to and including  $j$  is at most  $2^{i-2}$
- 

■ **Algorithm 4** **Insert**(job  $j$  with load vector  $\mathbf{p}(j) \in \mathbf{R}_{\geq 0}^d$ ).

- 
- 1: Run **SetPart**( $j$ ) to compute the partition  $i := \text{part}(j)$  the job  $j$  belongs to
  - 2: Job  $j$  chooses a uniformly random bin  $r(j) \in \mathcal{P}_i$
  - 3: **if**  $\max_k \sum_{j' \preceq j: r(j')=r(j)} p_k(j') \leq 1$ , i.e.,  $j$  can feasibly pack in bin  $r(j)$  **then**
  - 4:     Set assignment to be  $f(j) = r(j)$ , and add  $j$  to the set of regular jobs  $R(t)$
  - 5: **else**
  - 6:     Add  $j$  to spillover jobs  $S(t)$  and insert  $j$  to the spillover instance
  - 7: **end if**
- 

**Spillover Assignment Algorithm.** For the spillover instance, whenever our algorithm inserts a job  $j$  into  $S(t)$ , we simply *flatten* the vector into a 1-dimensional load  $\bar{p}_j = \sum_k p_k(j)$ , and feed it as an insertion into any  $O(1)$ -competitive  $O(1)$ -recourse fully-dynamic algorithm denoted by **1d-Alg** for bin packing, such as [13]. When the job is removed from  $S(t)$ , either by an adversarial deletion or by **Delete** algorithm when some job is being deleted, we treat it as a delete request to the **1d-Alg**.

#### 3.2 Analysis

We first show the competitive ratio analysis and then bound the recourse.

---

**Algorithm 5** Delete(job  $j$ ).
 

---

```

1: Remove  $j$  from the bin  $f(j)$  it belongs to, and from the corresponding set  $R(t)$  or  $S(t)$ 
   accordingly
2: for all jobs  $j' \succ j$  do
3:   Let  $i_{\text{old}} := \text{part}(j')$  denote its current partition
4:   Run  $\text{SetPart}(j')$  and let  $i_{\text{new}} := \text{part}(j')$  denote its new partition
5:   | if  $i_{\text{old}} \neq i_{\text{new}}$  then
6:     Remove  $j'$  from its appropriate set  $R(t)$  or  $S(t)$ , and from its current bin  $f(j')$ 
7:     Run  $\text{Insert}(j')$  ▷ Note that  $j'$  still retains its position in  $\prec$  ordering.
8:   | end if
9: end for

```

---

▷ **Claim 12.** For any job, the  $\text{part}(j)$  to which it is assigned is non-increasing over time.

*Proof.* The proof is straightforward because the  $\text{part}(j)$  of a job can change only when  $\text{SetPart}(j)$  is called. Moreover, the  $\text{SetPart}(j)$  only considers the total load of all active jobs which arrived before  $j$  to determine the partition. Since this is a non-increasing function of time, the claim follows. ◀

► **Lemma 13.** At any time step  $t$ , if there exists any job  $j \in A(t)$  with  $\text{part}(j) = i$ , then  $\text{Opt}(t) \geq 2^{i-3}$ .

*Proof.* Consider the latest time  $t' \leq t$  when  $j$ 's partition is updated by a call to  $\text{SetPart}(j)$ . Indeed, by definition of the **Delete** process, we know that there can be no deletions among jobs  $j' \preceq j$  after  $t'$ , i.e.,  $\{j' : j' \preceq_{t'} j\} = \{j' : j' \preceq_t j\}$ . Hence, we know that at time  $t'$  (and therefore at time  $t$  also), there exists some dimension  $k$  such that  $\sum_{j' \preceq_{t'} j} p_k(j') \geq 2^{i-3}$ , i.e., the dimension- $k$  load of all active jobs up to  $j$  at time  $t'$  exceeds  $2^{i-3}$ . Since none of these jobs  $j' \preceq_{t'} j$  are deleted between time steps  $t'$  and  $t$ , we get our desired lower bound on  $\text{Opt}(t)$ . ◀

► **Lemma 14.** For any job  $j$ , at all time steps, the probability that it belongs to  $S(t)$  is at most  $O(1/d)$ .

*Proof.* Consider the latest time  $t'$  prior to  $t$  when  $\text{Insert}(j)$  is invoked: this function determines whether  $j \in R(t')$  or  $j \in S(t')$ , and subsequently, this assignment has not changed till time step  $t$ . Now suppose  $j$  was placed in the spillover set  $S(t')$ , we show this happens with low probability. To this end, suppose  $j$  determined its  $\text{part}(j)$  to be  $i$ . This means that for the random choice  $r(j') \in \mathcal{P}_i$ , the total load on some dimension  $k$  of other jobs up to  $j' \prec_{t'} j$  in  $\mathcal{P}_i$  which chose the same bin  $r(j')$  is at least  $1 - \epsilon^2/d$ . However, since  $\text{part}(j) = i$ , we know that the total load of all jobs  $j' \prec_{t'} j$  on any dimension  $k$  is at most  $\text{load}_{t'}^k(\prec j) \leq 2^{i-2}$ . Since each of these jobs  $j'$  in the set  $\mathcal{P}_i \cap j'' \prec_{t'} j$  chose a uniformly random bin  $r(j')$  from one of  $2^{i-1}$  many bins in  $\mathcal{P}_i$ , the expected load in the particular machine  $r(j)$  is at most  $1/2$ . Since every individual job coordinate is at most  $\Theta(1/\log d)$ , a simple Chernoff bound establishes that the probability of job  $j$  being sent to the spillover set  $S(t)$  is at most  $O(1/d)$ . ◀

► **Lemma 15.** At all times  $t$ , the expected number of bins used by our algorithm is at most  $O(1)\text{Opt}(t)$ , where  $\text{Opt}(t)$  is the minimum number of bins needed to pack the set  $A(t)$  of active jobs.

*Proof.* The proof is a simple combination of Lemma 13, Lemma 14, along with the fact that the **1d-Alg** is  $O(1)$ -competitive for the 1-dimensional bin packing. Indeed, consider time  $t$  and let  $j_t$  denote the job which belongs to the largest partition, say  $i$ . Then the number

of bins used by the algorithm to serve regular jobs is at most  $\sum_{i' \leq i} 2^{i'-1} \leq 2^i$ . But we know that  $\text{Opt}(t) \geq 2^{i-3}$  from Lemma 13. Now for the spillover set  $S(t)$ , the expected total volume of jobs in  $S(t)$  (after flattening) is  $E[\sum_{j \in S(t)} \sum_k p_k(j)] \leq O(\frac{1}{d}) \sum_{j \in A(t)} \sum_k p_k(j)$  from Lemma 14. This in turn is at most  $O(1) \max_k \sum_{j \in A(t)} p_k(j) \leq O(1) \text{Opt}(t)$ . To complete the proof, note that the number of bins used by **1d-Alg** is at most  $O(1)$  times the total (flattened) volume of the instance  $S(t)$  due to its  $O(1)$ -competitiveness guarantees.  $\blacktriangleleft$

► **Lemma 16.** *The amortized recourse of the algorithm is at most  $O(\log n)$  per insert or delete update.*

**Proof.** Once a job is assigned a partition  $i$ , its bin choice  $r(j)$  does not change until the job migrates to a lower partition, which can happen  $O(\log n)$  times. Moreover, the total recourse within the **1d-Alg** is at most  $O(1)$  times the total number of update operations into the set  $S(t)$ , which is also  $O(\log n)T$  (where  $T$  is the length of the input sequence) for the same reason that a job can migrate between  $R(t)$  and  $S(t)$  only when its partition changes. This establishes the  $O(\log n)$  amortized recourse bound.  $\blacktriangleleft$

## 4 Deterministic Fully-Dynamic Vector Scheduling

In this section, we present a *deterministic algorithm* for **ONLINEVECTORSCHEDE** with  $O(\log d)$  competitive ratio and  $O(\log n + \log d \cdot \log(\log d \cdot \text{Opt}_{\max}/\text{Opt}_{\min}))$  recourse per job, where  $\text{Opt}_{\max}$  and  $\text{Opt}_{\min}$  are the maximum and minimum value of  $\text{Opt}$  through the existence of the job, and  $n$  is the maximum size of the active set during its existence.

At a high level, our algorithm maintains a potential function  $\Phi$  whose value depends on the current assignment of the active jobs  $A(t)$ . Moreover, each job additionally maintains a potential  $\Phi(j)$ , which is seen as its contribution to the overall potential. When a new job is inserted, we simply assign it to the machine which results in minimum potential, and  $\Phi(j)$  is precisely the increase in potential. Now, when a job  $j$  is deleted from the active set  $A(t)$ , then the potential of the other jobs which are inserted after  $j$  on the same machine will decrease. Hence, we run a *stabilize* routine, which *migrates any job* from its current machine (incurring current potential  $\Phi(j)$ ) to the end of any other machine (incurring what would be its new potential  $\Phi'(j)$  were it newly inserted into the other machine) if and only if  $\Phi'(j) < \Phi(j)/2$ .<sup>2</sup> In other words, we change the assignment of a job if there is a significant benefit for  $j$  w.r.t its potential to bound the recourse. In the scalar case, [2] uses similar techniques of stabilizing when jobs depart.

The potential  $\Phi$  is adapted from the scheduling algorithm in 1-dimension of [1]. Indeed, while many online algorithms use the exponential potential in the algorithms, this requires some normalization (along with the guess-and-double framework), which is not easy when there are deletions and  $\text{Opt}(t)$  is non-monotone. We, therefore, use a potential based on the sum of the  $L_q^q$  norm of the load vector on any machine, which avoids these issues. This potential has previously been used specifically for problems optimizing the  $L_q$  norm objective itself. Naively used, however, we would need  $q = \Theta(\log(md))$  for the  $L_q$  norm to approximate the load objective  $\max_{k \in [d]} \max_{i \in [m]} \text{load}_k(i)$ . This would in turn only yield a competitive ratio of  $\Theta(\log(md))$ . However, we can use stronger properties of the load vector (the fact that these machines are identical) to conclude that the sum of the dimensions' potentials for any two different machines is more or less similar (such ideas have also appeared in [17]), to obtain improved bounds of  $\Theta(\log d)$ .

<sup>2</sup> To get an improved recourse, we also use the condition  $\Phi(j)$  is at least  $q/n$  times the  $L_q^q$ -norm of the optimal maximum load for a suitable choice of  $q = \Theta(\log d)$ .

## 4.1 Useful Notation

At any time  $t$ , let  $f_t(\cdot)$  denote the assignment of jobs to machines maintained by the algorithm. We also maintain a total ordering  $\prec_t$  of all active jobs at all times (which unlike Section 2 is not based on the arrival order). Let  $n_t$  denote the size of  $A(t)$ . For any machine  $i$ , let  $\text{load}_t^k(i) = \sum_{j \in A(t): f_t(j)=i} p_k(j)$  denote the total load on dimension  $k$  at time  $t$ . Similarly, with a slight abuse of notation, let  $\text{load}_t^k(\prec_t j) = \sum_{j' \prec_t j: f_t(j')=f_t(j)} p_k(j')$  denote the load on dimension  $k$  for jobs occurring earlier than  $j$  in the ordering  $\prec_t$  which are scheduled on the same machine as  $j$ . Similarly let  $\text{load}_t^k(\preceq j) = \sum_{j' \preceq j: f_t(j')=f_t(j)} p_k(j')$  denote the same quantity, but including job  $j$ .

Let  $f_t^*(\cdot)$  denote an optimal assignment of jobs in  $A(t)$  to machines. For any machine  $i$ , let  $\text{optload}_t^k(i) = \sum_{j \in A(t): f_t^*(j)=i} p_k(j)$  denote the total load on dimension  $k$  at time  $t$  under  $f_t^*$ . Let  $\text{Opt}(t) = \max_i \max_k \text{optload}_t^k(i)$  denote the maximum load in the optimal assignment over all machines and dimensions.

**Total Potential and Job Potentials.** For a suitable choice of  $q > 1$ , we define a potential function  $\Phi_t = \sum_i \sum_k \text{load}_t^k(i)^q$ , which is the  $L_q^q$ -norm of the load vector. We decompose the potential as individual job potentials  $\Phi_t(j)$ , where  $\Phi_t(j) = \sum_k (\text{load}_t^k(\preceq_t j)^q - \text{load}_t^k(\prec_t j)^q)$  is the *contribution to the potential* due to  $j$ . It is easy to see that  $\Phi_t = \sum_{j \in A(t)} \Phi_t(j)$ . When time  $t$  is clear from the context, we will omit writing it for better readability. For example, we shall use  $f$ ,  $\text{load}^k(i)$ ,  $\prec$ , etc. to denote the respective terms at the current time step  $t$ .

Finally, let  $L(i)^q = \sum_k \text{load}^k(i)^q$  denote the  $L_q^q$ -norm of the load vectors of the assignment from the algorithm in machine  $i$ . Let  $L^*(i)^q = \sum_k \text{optload}^k(i)^q$  denote the  $L_q^q$ -norm of the load vector in the optimal assignment in machine  $i$  and let  $L^* = \max_i L^*(i)$ . For a cleaner presentation, we are assuming that we can calculate these quantities exactly, but our proofs carry through with an approximate lower bound also, with a slightly worse recourse. Moreover, note these quantities are a function of the active set  $A(t)$ , and we have just omitted the references to  $t$  since it is clear from the context.

## 4.2 Algorithm Definition

■ **Algorithm 6** Insert(job  $j$  with load vector  $\mathbf{p}(j) \in \mathbf{R}_{\geq 0}^d$ ).

- 
- 1: Set  $f(j)$  to be  $\text{argmin}_i \sum_k ((\text{load}^k(i) + p_k(j))^q - \text{load}^k(i)^q)$ .
  - 2: Update  $\prec$  by adding  $j$  as the last job in the ordering.
  - 3: Update  $\Phi$  and  $\Phi(j)$  as defined in Section 4.1.  $\Phi(j')$  for all  $j' \neq j$  remain unchanged.
- 

■ **Algorithm 7** Delete(job  $j$ ).

- 
- 1: Remove  $j$  from the machine  $f(j)$ , and from the ordering  $\prec$ , and update  $\Phi(j)$  for all  $j'$  s.t.  $f(j') = f(j)$ .
  - 2: Update  $\Phi$  and  $\Phi(j)$  as defined in Section 4.1.  $\Phi(j')$  for all  $j' \neq j$  remain unchanged.
  - 3: Run **Stabilize** sub-routine.
- 

In the next two subsections, we bound the competitive ratio and recourse of this algorithm.

■ **Algorithm 8** Stabilize.

- 
- 1: **while** there exists job  $j$ , machine  $i$  such that  $\Phi(j) > q(L^*)^q/n$  and  $\Phi(j) > 2 \sum_k ((\text{load}^k(i) + p_k(j))^q - \text{load}^k(i)^q)$  **do**
  - 2:   Remove  $j$  from its current machine  $i_{\text{orig}} = f(j)$  and place it in machine  $i$ , i.e., set  $f(j) = i$ .
  - 3:   Update  $\prec$  by removing  $j$  from its current position and adding it as the last job in the ordering.
  - 4:   Update  $\Phi(j)$  and  $\Phi(j')$  for all  $j'$  such that  $f(j') = i_{\text{orig}}$ .
  - 5: **end while**
- 

## 4.3 Analysis

### 4.3.1 Competitive Ratio

We know the potential for each job is bounded by the terms in Algorithm 8, as otherwise the job would be reassigned to other machines which might lead to a lower potential. We will use this to first bound the sum of the potentials of all the jobs, and then compare the loads of any two machines, to prove they are not too far apart.

► **Lemma 17.** *For a suitable choice of  $q$ , the algorithm in Section 4.2 maintains an assignment with the maximum load on any dimension at most  $O(\log d) \cdot \text{Opt}(t)$  at all times, where  $\text{Opt}(t)$  is the maximum load of an optimal assignment for the jobs  $A(t)$ .*

**Proof.** For the proof, let us fix a time  $t$ , and omit all references to the time step  $t$  unless necessary.

We first show that it is sufficient to bound the  $L_q$ -norm of the loads of the assignment the algorithm maintains against the  $L_q$ -norm of the optimal assignment when  $q = \log d$ .

► **Lemma 18.**  $\text{Opt}(t) = \Theta(L^*)$  when  $q = \log d$ .

**Proof.** Recall  $L^* = \max_i L^*(i)$  and let  $\text{Opt}(i) = \max_k \text{optload}^k(i)$ .

$$\begin{aligned}
 \log L^*(i) &= \frac{1}{q} \log \left( \sum_{k=1}^d \text{optload}^k(i)^q \right) \\
 &\leq \frac{1}{q} \log(d \cdot \text{Opt}(i)^q) && \text{(by definition of } \text{Opt}(i)) \\
 &\leq \frac{\log d}{q} + \log \text{Opt}(i) \\
 &\leq 1 + \log \text{Opt}(i)
 \end{aligned}$$

Hence,  $\text{Opt}(i) \leq L^*(i) \leq e \cdot \text{Opt}(i)$ . As  $\text{Opt}(t) = \max_i \text{Opt}(i)$ , we are done. ◀

Now we prove that the maximum load is  $O(\log d) \cdot L^*$ . We need the following lemma to be used later.

► **Lemma 19.** *[1]  $(l+r)^p \leq cl^p + (r(\frac{p}{\ln c} + 1))^p$  for any value of  $c, l, p, r > 0$ .*

This next lemma bounds the total potential of all the jobs, using the condition from Algorithm 8.

► **Lemma 20.**  $\sum_i \sum_{k=1}^d \text{load}^k(i)^q = O(m \cdot q^q (L^*)^q)$ .



The previous lemma suffices to get  $O(\log m)$  competitive ratio by setting  $q = \Theta(\log m)$ . However, we can go further, by actually showing that the sum of potentials on any two machines  $i$  and  $i'$  are comparable. This will help us get the improved  $\Theta(\log d)$  competitive ratio.

► **Lemma 21.**  $L(i) \leq O(qL^*)$  for all machines  $i$ .

We know  $\max_{i,k} \text{load}^k(j) \leq L(i)$  is always less than  $L(i)$  for the chosen value of  $i$ . From Lemma 21, the proof follows. ◀

### 4.3.2 Recourse

Now we will bound the number of times a job can be reassigned to another machine. Each time a job is reassigned, its potential reduces by half. Moreover, it cannot be reassigned after its potential drops below  $q(L^*)^q/n$  at any time  $t$ .

► **Lemma 22.** Any job  $j$  inserted at time  $t$  and deleted at time  $t'$  is reassigned to another machine  $O(q \log(\frac{qL_t^*}{L_{\min}^*}) + \log n_{\max})$  times where  $L_{\min}^* = \min_{r:t \leq r \leq t'} L_r^*$ ,  $L^* = L_t^*$ ,  $n_{\max} = \max_{r:t \leq r \leq t'} |A(r)|$ .

**Proof.** The maximum value of the potential for the job is bounded by

$$\begin{aligned}
\Phi_t(j) &= \sum_{k=1}^d (\text{load}^k(\preceq j)^q - \text{load}^k(\prec j)^q) \\
&\leq \sum_{k=1}^d q \cdot p_k(j) (\text{load}^k(\prec j) + p_k(j))^{q-1} \quad (\text{by convexity of } x^q \text{ when } x > 0 \text{ for } q > 1) \\
&\leq \sum_{k=1}^d q \cdot p_k(j) (cqL_t^*)^{q-1} \quad (\text{from Lemma 20, for some } c) \\
&\leq q(cqL_t^*)^{q-1} \cdot \sum_{k=1}^d p_k(j) \\
&\leq q(cqL_t^*)^{q-1} \cdot dL_t^* \\
&\leq d(cqL_t^*)^q
\end{aligned}$$

Therefore, the number of times it is reassigned is bounded by  $O(\log(\frac{cqL_t^*}{L_{\min}^*})^q \cdot \frac{dn_{\max}}{q}) = O(\log n_{\max} + q \cdot \log(qL_t^*/L_{\min}^*))$ . ◀

## 5 Conclusions and Future Directions

In this paper, we have proposed the first algorithms for vector versions of the fully-dynamic scheduling and bin packing variants, but there are many interesting directions that need investigation. Two immediate open questions are (i) an  $O(1)$  amortized recourse *deterministic algorithm* for vector scheduling (with  $O(\log d/\log \log d)$  competitive ratio), and (ii) an  $O(1)$  amortized recourse algorithm for fully dynamic vector bin packing under the small job assumption (with  $O(1)$  competitive ratio). Extension of fully-dynamic vector scheduling beyond identical machines to related or unrelated machines is a natural next step. The potential function based algorithm of [17] that we build on in Section 4 gives an  $O(\log m + \log d)$  competitive ratio for unrelated machines in the arrival only model, so we expect similar guarantees should be possible for the fully-dynamic setting as well. Finally, moving closer to applications, algorithms for semi-adversarial instances such as random order or models with predictions (e.g., [7]) would be interesting and impactful as well.

## References

- 1 B. Awerbuch, Y. Azar, E. Grove, M. Kao, P. Krishnan, and J. Vitter. Load balancing in the lp norm. In *FOCS 1995*, 1995.
- 2 Baruch Awerbuch, Yossi Azar, Serge Plotkin, and Orli Waarts. Competitive routing of virtual circuits with unknown duration. *J. Comput. Syst. Sci.*, 62(3):385–397, May 2001. doi:10.1006/jcss.1999.1662.
- 3 Yossi Azar, Ilan Reuven Cohen, Amos Fiat, and Alan Roytman. Packing small vectors. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '16*, pages 1511–1525, USA, 2016. Society for Industrial and Applied Mathematics.
- 4 Yossi Azar, Ilan Reuven Cohen, Seny Kamara, and Bruce Shepherd. Tight bounds for online vector bin packing. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing, STOC '13*, pages 961–970, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2488608.2488730.
- 5 Nikhil Bansal, Alberto Caprara, and Maxim Sviridenko. A new approximation method for set covering problems, with applications to multidimensional bin packing. *SIAM J. Comput.*, 39(4):1256–1278, 2009.
- 6 Nikhil Bansal, Marek Eliáš, and Arindam Khan. Improved approximation for vector bin packing. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 1561–1579, 2016.
- 7 Niv Buchbinder, Yaron Fairstein, Konstantina Mellou, Ishai Menache, and Joseph (Seffi) Naor. Online virtual machine allocation with lifetime and load predictions. In *Abstract Proceedings of the 2021 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '21*, pages 9–10, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3410220.3456278.
- 8 Chandra Chekuri and Sanjeev Khanna. On multidimensional packing problems. *SIAM J. Comput.*, 33(4):837–851, 2004.
- 9 Henrik I. Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. Approximation and online algorithms for multidimensional bin packing: A survey. *Comput. Sci. Rev.*, 24:63–79, 2017.
- 10 Edward G Coffman, Jr, Michael R Garey, and David S Johnson. Dynamic bin packing. *SIAM Journal on Computing*, 12(2):227–258, 1983.
- 11 Graham Cormode and Pavel Veselý. Streaming algorithms for bin packing and vector scheduling. In Evripidis Bampis and Nicole Megow, editors, *Approximation and Online Algorithms - 17th International Workshop, WAOA 2019, Munich, Germany, September 12-13, 2019, Revised Selected Papers*, volume 11926 of *Lecture Notes in Computer Science*, pages 72–88. Springer, 2019.
- 12 Wenceslas Fernandez de la Vega and George S. Lueker. Bin packing can be solved within  $1+\epsilon$  in linear time. *Comb.*, 1(4):349–355, 1981.
- 13 Björn Feldkord, Matthias Feldotto, Anupam Gupta, Guru Guruganesh, Amit Kumar, Sören Riechers, and David Wajc. Fully-Dynamic Bin Packing with Little Repacking. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 51:1–51:24, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2018.51.
- 14 M. R. Garey, Ronald L. Graham, David S. Johnson, and Andrew Chi-Chih Yao. Resource constrained scheduling as generalized bin packing. *J. Comb. Theory, Ser. A*, 21(3):257–298, 1976.
- 15 David G. Harris and Aravind Srinivasan. The Moser-Tardos framework with partial resampling. *J. ACM*, 66(5):36:1–36:45, 2019.
- 16 Sungjin Im, Nathaniel Kell, Janardhan Kulkarni, and Debmalya Panigrahi. Tight bounds for online vector scheduling. *SIAM J. Comput.*, 48(1):93–121, 2019.

- 17 Adam Meyerson, Alan Roytman, and Brian Tagiku. Online multidimensional load balancing. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 287–302, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 18 Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- 19 Sai Sandeep. Almost optimal inapproximability of multidimensional packing problems. *CoRR*, abs/2101.02854, 2021. [arXiv:2101.02854](https://arxiv.org/abs/2101.02854).
- 20 Jeffery Westbrook. Load balancing for response time. In *J. Algorithms*, pages 355–368, 1994.