Scheduling Jobs with Random Resource Requirements in Computing Clusters

Konstantinos Psychas, Javad Ghaderi
Electrical Engineering, Columbia University, New York, NY, USA
Email: {kpsychas, jghaderi}@ee.columbia.edu

***Abstract*—We consider a natural scheduling problem which arises in many distributed computing frameworks. Jobs with diverse resource requirements (e.g. memory requirements) arrive over time and must be served by a cluster of servers, each with a finite resource capacity. To improve throughput and delay, the scheduler can pack as many jobs as possible in the servers subject to their capacity constraints. Motivated by the ever-increasing complexity of workloads in shared clusters, we consider a setting where the jobs' resource requirements belong to a very large number of diverse types or, in the extreme, even infinitely many types, e.g. when resource requirements are drawn from an *unknown* distribution over a continuous support. The application of classical scheduling approaches that crucially rely on a predefined finite set of types is discouraging in this high (or infinite) dimensional setting. We first characterize a fundamental limit on the maximum throughput in such setting, and then develop oblivious scheduling algorithms that have *low complexity* and can achieve *at least* 1/2 and 2/3 of the maximum throughput, *without the knowledge of traffic or resource requirement distribution*. Extensive simulation results, using both synthetic and real traffic traces, are presented to verify the performance of our algorithms.**

***Index Terms*—Scheduling Algorithms, Stability, Queues, Knapsack, Data Centers**

I. Introduction

Distributed computing frameworks (e.g., MapReduce [1], Spark [2], Hive [3]) have enabled processing of very large data sets across a cluster of servers. The processing is typically done by executing a set of jobs or tasks in the servers. A key component of such systems is the resource manager (*scheduler*) that assigns incoming jobs to servers and reserves the requested resources (e.g. CPU, memory) on the servers for running jobs. For example, in Hadoop [1], the resource manager reserves the requested resources, by launching *resource containers* in servers. Jobs of various applications can arrive to the cluster, which often have very diverse resource requirements. Hence, to improve throughput and delay, a scheduler should pack as many jobs (containers) as possible in the servers, while retaining their resource requirements and not exceeding server's capacities.

A salient feature of resource demand is that it is hard to predict and cannot be easily classified into a small or moderate number of resource profiles or "*types*". This is amplified by the increasing complexity of workloads, i.e., from traditional batch jobs, to queries, graph processing, streaming, machine learning jobs, etc., that rely on multiple computation frameworks, and all need to share *the same cluster*. For example, Figure 1 shows the statistics of memory and CPU resource requirement requested by jobs in a Google cluster [4], over the first day in the trace. If jobs were to be divided into types according to their memory requirement alone, there would be more than 700 types. Moreover, the statistics change over time and these types are not sufficient to model all the job requirements in a month, which are more than 1500. We can make a similar observation for CPU requirements, which take more than 400 discrete types. Analyzing the joint CPU and memory requirements, there would be more than $10,000$ distinct types. Building a low-complexity scheduler that can provide high performance in such a high-dimensional regime is extremely challenging, as learning the demand for all types is infeasible, and finding the optimal packing of jobs in servers, even when the demand is known, is a hard combinatorial problem (related to *Bin Packing* and *Knapsack* problems [5]).

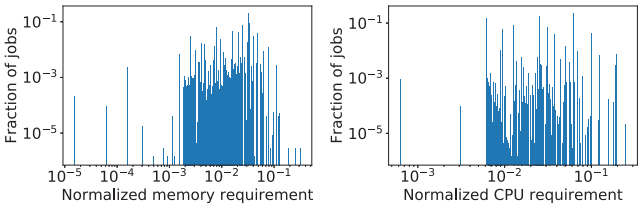


Fig. 1: There are more than 700 discrete memory requirements and 400 discrete CPU requirements in the tasks submitted to a Google cluster during a day.

Despite the vast literature on scheduling algorithms, their theoretical study in such high-dimensional setting is very limited. The majority of the past work relies on a crucial assumption that there is a predefined finite set of discrete types, e.g. [6], [7], [8], [9], [10], [11]. Although we can consider every possible resource profile as a type, the number of such types could be formidably large. The application of scheduling algorithms, even with polynomial complexity in the number of types, is discouraging in such setting. A natural solution could be to divide the resource requests into a smaller number of types. Such a scheduler can be strictly suboptimal, since, as a result of mapping to a smaller number of types, jobs may underutilize or overutilize the resource compared to what they actually require. Moreover, in the *absence of any prior knowledge about the resource demand statistics*, it is not clear

This research was supported by NSF Grants CNS-1652115 and CNS-1717867.

how the partitioning of the resource axis into a small number of types should be actually done.

Our work fulfills one of the key deficiencies of the past work in the modeling and analysis of scheduling algorithms for distributed server systems. Our model allows a very large or, in the extreme case, even *infinite* number of job types, i.e., when the jobs' resource requirements follow a probability distribution over a continuous support. To the best of our knowledge, there is no past work on characterizing the optimal throughput and what can be achieved when there are no discrete job types. Our goal is to characterize this throughput and design algorithms that: (1) have low complexity, and (2) can provide provable throughput guarantees *without* the knowledge of the traffic or the resource requirement statistics.

A. Related Work

Existing algorithms for scheduling jobs in distributed computing platforms can be organized in two categories.

In the first category, we have schedulers with throughput guarantees, e.g., [6], [8], [9], [10], [11]. They work under the assumption that there is a finite number of discrete job types. This assumption naturally lends itself to *MaxWeight* algorithms [12], where each server schedules jobs according to a maximum weight configuration chosen from a finite set of configurations. The number of configurations however grows exponentially large with the number of types, making the application of these algorithms discouraging in practice. Further, their technique *cannot* be applied to our setting which can include an infinite number of job types.

In the second category, we have algorithms that do not provide any throughput guarantees, but perform well empirically or focus on other performance metrics such as fairness and makespan. These algorithms include slot-based schedulers that divide servers into a predefined number of slots for placing tasks [13], [14], resource packing approaches such as [15], [16], fair resource sharing approaches such as [17], [18], and Hadoop's default schedulers such as FIFO [19], Fair scheduler [20], and Capacity scheduler [21]. The methodology in our work goes beyond the approaches that assume a finite number of job types and hence can be potentially used to analyze the performance of algorithms in this category.

There is also literature on classical bin packing problem [22], where given a list of objects of various sizes, and an infinite number of unit-capacity bins, the goal is to use the minimum number of bins to pack the objects. Many algorithms have been proposed for this problem with approximation ratios for the optimal number of bins or waste, e.g. [23], [24], [25]. There is also work in a setting of bin packing with queues [26], [27], [28], under the model that an empty bin arrives at each time, then some jobs from the queue are packed in the bin at that time, and the bin cannot be reused in future. Our model is *fundamentally* different from these lines of work, as the number of servers (bins) in our setting is fixed and we need to reuse the servers to schedule further jobs from the queue, when jobs depart from servers.

B. Main Contributions

Our main contributions can be summarized as follows:

1. **Characterization of Maximum Achievable Throughput.** We characterize the maximum throughput (*maximum supportable workload*) that can be theoretically achieved by any scheduling algorithm in the setting that the jobs' resource requirements follow a general probability distribution F_R over possibly infinitely many job types. The construction of optimal schedulers to approach this maximum throughput relies on a careful partition of jobs into sufficiently large number of types, using the complete knowledge of the resource probability distribution F_R.
2. **Oblivious Scheduling Algorithms.** We introduce scheduling algorithms based on "*Best-Fit*" packing and "*universal partitioning*" of resource requirements into types, *without* the knowledge of the resource probability distribution F_R. The algorithms have low complexity and can provably achieve at least $1/2$ and $2/3$ of the maximum throughput, respectively. Further, we show that $2/3$ is tight in the sense that no oblivious scheduling algorithm, that maps the resource requirements into a finite number of types, can achieve better than $2/3$ of the maximum throughput for all general resource distributions F_R.
3. **Empirical Evaluation.** We evaluate the throughput and queueing delay performance of all algorithms empirically using both synthetic and real traffic traces.

II. System Model and Definitions

Cluster Model: We consider a collection of L servers denoted by the set $\mathcal{L}$. For simplicity, we consider a single resource (e.g. memory) and assume that the servers have the same resource capacity. While job resource requirements are in general multi-dimensional (e.g. CPU, memory), it has been observed that memory is typically the bottleneck resource [21], [29]. Without loss of generality, we assume that each server's capacity is normalized to one.

Job Model: Jobs arrive over time, and the i-th job, $i = 1, 2, \cdots$, requires an amount R_i of the (normalized) resource for the duration of its service. The resource requirements $R_1, R_2, \cdots$ are i.i.d. random variables with a *general* cdf (cumulative distribution function) $F_R(\cdot) : (0, 1] \to [0, 1]$, with average $\bar{R} = \mathbb{E}(R)$. Note that each job should be served by one server and its resource requirement *cannot be fragmented* among multiple servers. In the rest of the paper, we use the terms job size and job resource requirement interchangeably.

Queueing Model: We assume time is divided into time slots $t = 0, 1, \cdots$. At the beginning of each time slot t, a set $\mathcal{A}(t)$ of jobs arrive to the system. We use $A(t)$ to denote the cardinality of $\mathcal{A}(t)$. The process $A(t)$, $t = 0, 1, \cdots$, is assumed to be i.i.d. with a finite mean $\mathbb{E}[A(t)] = \lambda$ and a finite second moment.

There is a queue $\mathcal{Q}(t)$ that contains the jobs that have arrived up to time slot t and have not been served by any servers yet. At each time slot, the scheduler can select a set of jobs $\mathcal{D}(t)$ from $\mathcal{Q}(t)$ and place each job in a server that

has enough available resource to accommodate it. Specifically, define $\mathcal{H}(t) = (\mathcal{H}_\ell(t),\ \ell \in \mathcal{L})$, where $\mathcal{H}_\ell(t)$ is the set of existing jobs in server ℓ at time t. At any time, the total size of the jobs packed in server ℓ *cannot* exceed its capacity, i.e.,

$$\sum_{j\in\mathcal{H}_\ell(t)} R_j \le 1,\ \forall \ell \in \mathcal{L},\ t = 0, 1, \cdots. \tag{1}$$

Note that jobs may be scheduled out of the order that they arrived, depending on the resource availability of servers. Let $D(t)$ denote the cardinality of $\mathcal{D}(t)$ and $Q(t)$ denote the cardinality of $\mathcal{Q}(t)$ (the number of jobs in the queue). Then the queue $\mathcal{Q}(t)$ and its size $Q(t)$ evolve as

$$\mathcal{Q}(t+1) = \mathcal{Q}(t) \cup \mathcal{A}(t) - \mathcal{D}(t), \tag{2}$$
$$Q(t+1) = Q(t) + A(t) - D(t). \tag{3}$$

Once a job is placed in a server, it completes its service after a geometrically distributed amount of time with mean $1/\mu$, after which it releases its reserved resource. This assumption is made to simplify the analysis, and the results can be extended to more general service time distributions (see Section VIII for a discussion).

Stability and Maximum Supportable Workload: The system state is given by $(\mathcal{Q}(t), \mathcal{H}(t))$ which evolves as a Markov process over an *uncountably infinite state space*[1]. We investigate the stability of the system in terms of the average queue size, i.e., the system is called stable if $\limsup_t \mathbb{E}[Q(t)] < \infty$. Given a job size distribution F_R, a workload $\rho := \lambda/\mu$ is called supportable if there exists a scheduling policy that can stabilize the system for the job arrival rate λ and the mean service duration $1/\mu$.

Maximum supportable workload is a workload $\rho^\star$ such that any $\rho < \rho^\star$ can be stabilized by some scheduling policy, which possibly uses the knowledge of the job size distribution F_R, but no $\rho > \rho^\star$ can be stabilized by any scheduling policy.

III. Characterization of Maximum Supportable Workload

In this section, we provide a framework to characterize the maximum supportable workload $\rho^\star$ given a job resource distribution F_R. We start with an overview of the results for a system with a finite set of discrete job types.

A. Finite-type System

It is easy to characterize the maximum supportable workload when jobs belong to a finite set of discrete types. In this case, it is well known that the supportable workload region is the sum of convex hull of *feasible configurations* of servers, e.g. [6], [8], [9], [10], [11], which are defined as follows.

Definition 1 (Feasible configuration)**.** *Suppose there is a finite set of J job types, with job sizes $r_1, \cdots, r_J$. An integer-valued vector $\mathbf{k} = (k_1, \cdots, k_J)$ is a feasible configuration for a server if it is possible to simultaneously pack k_1 jobs of of type 1, k_2 jobs of type 2, ..., and k_J jobs of type J in the server, without exceeding its capacity. Assuming normalized server's capacity, any feasible configuration $\mathbf{k}$ must therefore satisfy $\sum_{j=1}^J k_j r_j \le 1$, $k_j \in \mathbb{Z}_+$, $j = 1, \cdots, J$. We use $\overline{\mathcal{K}}$ to denote the (finite) set of all feasible configurations.*

Define $P_j \triangleq \mathbb{P}(R = r_j)$ to be the probability that the size of an arriving job is r_j, $\boldsymbol{P} = (P_1, \cdots, P_J)$, and the workload $\rho = \lambda/\mu$. The maximum supportable workload $\rho^\star$ is

$$\rho^\star = \sup\left\{\rho \in \mathbb{R}_+ : \rho\boldsymbol{P} < \sum_{\ell\in\mathcal{L}} \mathbf{x}^\ell, \mathbf{x}^\ell \in \mathrm{Conv}(\overline{\mathcal{K}}), \ell \in \mathcal{L}\right\} \tag{4}$$

where $\mathrm{Conv}(\cdot)$ is the convex hull operator, and the vector inequality is component-wise. Also $\sup$ (or $\inf$) denotes *supremum* (or *infimum*). Hence any $\rho < \rho^\star$ is supportable by some scheduling algorithm, while no $\rho > \rho^\star$ can be supported by any scheduling algorithm.

The optimal or near-optimal scheduling policies then basically follow the well-known *MaxWeight* algorithm [12]. Let $Q_j(t)$ be the number of type-j jobs waiting in queue at time t. At any time t for each server ℓ, the algorithm maintains a feasible configuration $\mathbf{k}(t)$ that has the "maximum weight" [8], [9] (or a fraction of the maximum weight [11]), among all the feasible configurations $\overline{\mathcal{K}}$. The weight of a configuration is formally defined below.

Definition 2 (Weight of a configuration)**.** *Given a queue size vector $\boldsymbol{Q} = (Q_1, \cdots, Q_J)$, the weight of a feasible configuration $\mathbf{k} = (k_1, \cdots, k_J)$ is defined as the inner product*

$$\langle \mathbf{k}, \mathbf{Q}\rangle = \textstyle\sum_{j=1}^J k_j Q_j. \tag{5}$$

B. Infinite-type System

In general, the support of the job size distribution F_R can span an infinite number of types (e.g., F_R can be a continuous function over $(0, 1]$). We introduce the notion of virtual queue which is used to characterize the supportable workload for any general distribution F_R.

Definition 3 (Virtual queues (VQs))**.** *Define a partition X of interval $(0, 1]$ as a finite collection of disjoint subsets $X_j \subset (0, 1]$, $j = 1, \cdots, J$, such that $\cup_{j=1}^J X_j = (0, 1]$. Let $P_j = \mathbb{P}(R \in X_j)$ be the probability that resource requirement of an arriving job belongs to X_j, in which case we refer to it as a type-j job. For each type j, we consider a virtual queue $\mathrm{VQ_j}$ which contains the type-j jobs waiting in the queue for service.*

Under this definition, it is not clear what configurations are feasible, since the jobs in the same virtual queue can have different sizes, even though they are called of the same type. Hence we make the following definition.

Definition 4 (Rounded VQs)**.** *We call* VQs *"upper-rounded* VQs*", if the sizes of type-j jobs are assumed to be $r_j = \sup X_j$, $j = 1, \cdots, J$. Similarly, we call them "lower-rounded* VQs*", if the sizes of type-j jobs are assumed to be $r_j = \inf X_j$, $j = 1, \cdots, J$.*

[1]The state space can be equivalently represented in a complete separable metric space, see our technical report [30] for details.

Given a partition X, let $\overline{\rho}^\star(X)$ and $\underline{\rho}^\star(X)$ be respectively the maximum λ/μ under which the system with upper-rounded virtual queues and the system with the lower-rounded virtual queues can be stabilized. Let also $\overline{\rho}^\star = \sup_X \overline{\rho}^\star(X)$ and $\underline{\rho}^\star = \inf_X \underline{\rho}^\star(X)$ where the supremum and infimum are over all possible partitions of interval $(0, 1]$. Next theorem states the result of existence of maximum supportable workload.

Theorem 1. *Consider any general (continuous or discontinuous) probability distribution of job sizes with cdf $F_R(\cdot)$. Then there exists a unique $\rho^\star$ such that $\overline{\rho}^\star = \underline{\rho}^\star = \rho^\star$. Further, given any $\rho < \rho^\star$, there is a partition X such that the associated upper-rounded virtual queueing system (and hence the original system) can be stabilized.*

The proof of Theorem 1 has two steps. First, we show that $\overline{\rho}^\star(X) \leq \rho^\star \leq \underline{\rho}^\star(X)$ for any partition X. Second, we construct a sequence of partitions, that depend on the job size distribution F_R, and become increasingly finer, such that the difference between the two bounds vanishes in the limit. The proof details can be found in our technical report [30].

Theorem 1 implies that there is a way of mapping the job sizes to a finite number of types using partitions, such that by using finite-type scheduling algorithms, the achievable workload approaches the optimal workload as partitions become finer. However, the construction of the partition crucially *relies on the knowledge of the job size distribution F_R*, which may not be readily available in practice. Further, the number of feasible configurations *grows exponentially* large as the number of subsets in the partition increases, which prevents efficient implementation of discrete type scheduling policies (e.g. MaxWeight) in practice.

Next, we focus on low-complexity scheduling algorithms that *do not* assume the knowledge of F_R a priori, and can provide a fraction of the maximum supportable workload $\rho^\star$.

IV. Best-Fit Based Scheduling

The *Best-Fit* algorithm was first introduced as a heuristic for *Bin Packing* problem [22]: given a list of objects of various sizes, we are asked to pack them into bins of unit capacity so as to minimize the number of bins used. Under Best-Fit, the objects are processed one by one and each object is placed in the "tightest" bin (with the least residual capacity) that can accommodate the object, otherwise a new bin is used. Theoretical guarantees of Best-Fit in terms of approximation ratio have been extensively studied under discrete and continuous object size distributions [23], [24], [25].

There are several *fundamental* differences between the classical bin packing problem and our problem. In the bin packing problem, there is an infinite number of bins available and once an object is placed in a bin, it remains in the bin forever, while in our setting, the number of bins (the equivalent of servers) is fixed, and bins have to be reused to serve new objects from the queues as objects depart from the bins, and new objects arrive to the queue. Next, we describe how Best-Fit (*BF*) can be adapted for job scheduling in our setting.

A. BF-J/S Scheduling Algorithm

Consider the following two adaptations of Best-Fit (BF) for job scheduling:

- **BF-J** (*Best-Fit from Job's perspective*):
  List the jobs in the queue in an arbitrary order (e.g. according to their arrival times). Starting from the first job, each job is placed in the server with the "least residual capacity" among the servers that can accommodate it, if possible, otherwise the job remains in the queue.
- **BF-S** (*Best-Fit from Server's perspective*):
  List servers in an arbitrary order (e.g. according to their index). Starting from the first server, each server is filled iteratively by choosing the "largest-size job" in the queue that can fit in the server, until no more jobs can fit.

BF-J and BF-S need to be performed in every time slot. Under both algorithms, observe that no further job from the queue can be added in any of the servers. However, these algorithms are not computationally efficient as they both make many redundant searches over the jobs in the queue or over the servers, when there are no new job arrivals to the queue or there are no job departures from some servers. Combining both adaptations, we describe the algorithm below which is computationally more efficient.

- **BF-J/S** (*Best-Fit from Job's and Server's perspectives*):
  It consists of two steps:

1) Perform BF-S only over the list of servers that had job departures during the previous time slot. Hence, some jobs that have not been scheduled in the previous time slot or some of newly arrived jobs are scheduled in servers.
2) Perform BF-J only over the list of newly arrived jobs that have not been scheduled in the first step.

B. Throughput Guarantee

The following theorem characterizes the maximum supportable workload under BF-J/S.

Theorem 2. *Suppose any job has a minimum size u. Algorithm BF-J/S can achieve at least $\frac{1}{2}$ of the maximum supportable workload $\rho^\star$, for any $u > 0$.*

Proof Overview. The proof uses Lyapunov analysis for Markov chain $(\mathcal{Q}(t), \mathcal{H}(t))$ whose state includes the jobs in queues and servers and their sizes. The Markov chain can be equivalently represented in a Polish space and we prove its positive recurrence, by using Theorem 1 of [31] and properties of BF-J/S. We use a Lyapunov function which is the sum of sizes of all jobs in the system at time t. Given that jobs have a minimum size, keeping the total size bounded implies the number of jobs is also bounded.

The key argument in the proof is that by using BF-J/S as described, all servers operate in more than "half full", most of the time, when the total size of jobs in the queue becomes large. To prove this, we consider two possible cases:

- *The total size of jobs in queue with size $\leq \frac{1}{2}$ is large:*
  In this case, these jobs will be scheduled greedily whenever the server is more than half empty. Hence, the server will

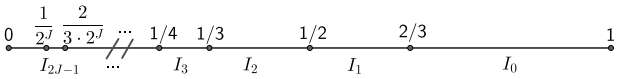


Fig. 2: Partition I of interval $(1/2^J, 1]$ based on (6).

always become more than half full until there are no such jobs in the queue.

- *The total size of jobs in queue with size $> \frac{1}{2}$ is large:* If at time slot t, a job in server is not completed, it will complete its service within the next time slot with probability μ, independently of the other jobs in the server. Given the minimum job size, the number of jobs in a server is bounded so it will certainly empty in a finite time. Once this happens, jobs will be scheduled starting from the largest-size one, and the server will remain more than half full, as long as there is a job of size more than $1/2$ to replace it. This step is true because of the way Best-Fit works and *does not* hold for other bin packing algorithms like First-Fit.

The proof details can be found in [30]. □

V. Partition Based Scheduling

BF-J/S demonstrated an algorithm that can achieve at least half of the maximum workload $\rho^\star$, without relying on any partitioning of jobs into types. In this section, we propose partition based scheduling algorithms that can provably achieve a larger fraction of the maximum workload $\rho^\star$, using a *universal partitioning* into a small number of types, without the knowledge of job size distribution F_R.

A. Universal Partition and Associated Virtual Queues

Consider a partition of the interval $(1/2^J, 1]$ into the following $2J$ sub-intervals:

$$\begin{aligned} I_{2m} &= \Big(\frac{2}{3}\frac{1}{2^m}, \frac{1}{2^m}\Big],\ m = 0, \cdots, J-1 \\ I_{2m+1} &= \Big(\frac{1}{2}\frac{1}{2^m}, \frac{2}{3}\frac{1}{2^m}\Big],\ m = 0, \cdots, J-1. \end{aligned} \tag{6}$$

We refer to this partition as partition I, where $J > 1$ is a fixed parameter to be determined shortly. The odd and even sub-intervals in I are geometrically shrinking. Figure 2 gives a visualization of this partition.

Jobs in queue are divided among virtual queues (Definition 3) according to partition I. Specifically, when the size of a job falls in the sub-interval I_j, $j = 0, \cdots, 2J-1$, we say this job is of type j and it is placed in a virtual queue VQ_j, *without rounding its size*. Moreover, jobs whose sizes fall in $(0, 1/2^J]$ are placed in the last virtual queue VQ_{2J-1}, and their sizes are rounded up to $1/2^J$.

We use $Q_j(t)$ to denote the size (cardinality) of VQ_j at time t and use $\mathbf{Q}(t)$ to denote the vector of all VQ sizes.

B. VQS (Virtual Queue Scheduling) Algorithm

To describe the VQS algorithm, we define the following reduced set of configurations which are feasible for the system of upper-rounded VQs (Definition 4)

Definition 5 (Reduced feasible configuration set)**.** *The reduced feasible configuration set, denoted by $\mathcal{K}_{RED}^{(J)}$, consists of the following $4J-4$ configurations:*

$$\begin{aligned} 2^m \mathbf{e}_{2m}, &\quad m = 0, \cdots, J-1 \\ 3 \cdot 2^{m-1} \mathbf{e}_{2m+1}, &\quad m = 1, \cdots, J-1 \\ \mathbf{e}_1 + \lfloor 2^m/3 \rfloor \mathbf{e}_{2m}, &\quad m = 2, \cdots, J-1 \\ \mathbf{e}_1 + 2^{m-1} \mathbf{e}_{2m+1}, &\quad m = 1, \cdots, J-1 \end{aligned} \tag{7}$$

where $\mathbf{e}_j \in \mathbb{Z}^{2J}$ denotes the basis vector with a single job of type j, $j = 0, \cdots, 2J-1$, and zero jobs of any other types.

Note that each configuration $\mathbf{k} = (k_0, \cdots, k_{2J-1}) \in \mathcal{K}_{RED}^{(J)}$ either contains jobs from only one VQ_j, $j = 0, \cdots, 2J-1$, or contains jobs from VQ_1 and one other VQ_j.

The "*VQS algorithm*" consists of two steps: (1) setting active configuration, and (2) job scheduling using the active configuration:

1. *Setting active configuration*:
   Under VQS, every server $\ell \in \mathcal{L}$ has an *active configuration* $\mathbf{k}^\ell(t) \in \mathcal{K}_{RED}^{(J)}$ which is renewed only when the server becomes empty. Suppose time slot τ_i^ℓ is the i-th time that server ℓ is empty (i.e., it has been empty or all its jobs depart during this time slot). At this time, the configuration of server ℓ is set to the max weight configuration among the configurations of $\mathcal{K}_{RED}^{(J)}$ (Definitions 2 and 5), i.e.,

$$\mathbf{k}^\star(\tau_i^\ell) = \arg\max_{\mathbf{k} \in \mathcal{K}_{RED}^{(J)}} \langle \mathbf{k}, \mathbf{Q}(\tau_i^\ell) \rangle = \arg\max_{\mathbf{k} \in \mathcal{K}_{RED}^{(J)}} \sum_{j=0}^{2J-1} k_j Q_j. \tag{8}$$

   The active configuration remains fixed until the next time τ_{i+1}^ℓ that the server becomes empty gain, i.e.,

$$\mathbf{k}^\ell(t) = \mathbf{k}^\star(\tau_i^\ell),\ \tau_i^\ell \le t < \tau_{i+1}^\ell. \tag{9}$$

2. *Job scheduling:*
   Suppose the active configuration of server ℓ at time t is $\mathbf{k} \in \mathcal{K}_{RED}^{(J)}$. Then the server schedules jobs as follows:
   (i) If $k_1 = 1$, the server reserves $2/3$ of its capacity for serving jobs from VQ_1, so it can serve at most one job of type 1 at any time. If there is no such job in the server already, it schedules one from VQ_1.
   (ii) Any configuration $\mathbf{k} \in \mathcal{K}_{RED}^{(J)}$ has at most one $k_j > 0$ other than k_1. The server will schedule jobs from the corresponding VQ_j, starting from the head-of-the-line job in VQ_j, until no more jobs can fit in the server. The actual number of jobs scheduled from VQ_j in the server could be more than k_j depending on their actual sizes.

Remark 1. The reason for choosing times τ_i^ℓ to renew the configuration of server ℓ is to *avoid possible preemption* of existing jobs in server (similar to [6], [9]). Also note that active configurations in $\mathcal{K}_{RED}^{(J)}$ are based on upper-rounded VQs. Since jobs *are not* actually rounded in VQs, the algorithm can schedule more jobs than what specified in the configuration.

C. Throughput Guarantee

The VQS algorithm can provide a stronger throughput guarantee than BF-J/S. A key step to establish the throughput guarantee is related to the property of configurations in the set $\mathcal{K}_{RED}^{(J)}$, which is stated below.

Proposition 1. *Consider any partition X which is a refinement of partition I, i.e., any interval in X is contained in an interval I_j in (6). Given any set of jobs with sizes in $(1/2^J, 1]$ in the queue, let $\boldsymbol{Q}$ and $\mathbf{Q}^{(X)}$ be the corresponding vector of* VQ *sizes under partition I and partition X. Then there is a configuration $\mathbf{k} \in \mathcal{K}_{RED}^{(J)}$ such that*

$$\langle \mathbf{k}, \mathbf{Q} \rangle \geq \frac{2}{3} \langle \mathbf{k}^{(X)}, \mathbf{Q}^{(X)} \rangle, \ \forall \mathbf{k}^{(X)} \in \mathcal{K}^{(X)}, \tag{10}$$

where $\mathcal{K}^{(X)}$ is the set of "all" feasible configurations based on upper-rounded VQ*s for partition X.*

Proof. Suppose X is a partition of $(1/2^J, 1]$ into N sub-intervals $(\xi_{i-1}, \xi_i]$, $i = 1, \cdots, N$. Given the proposition's assumption, we can define sets Z_j, $j = 0, \cdots, 2J-1$, such that $i \in Z_j$ iff $\xi_i \in I_j$. Any job in $\text{VQ}_i^{(X)}$, $i \in Z_j$, under partition X, belongs to VQ_j under partition I, therefore $\sum_{i \in Z_j} Q_i^{(X)} = Q_j$.

Let $\langle \mathbf{k}^{(X)}, \mathbf{Q}^{(X)} \rangle = U$. Note that in any feasible configuration $\mathbf{k}^{(X)} \in \mathcal{K}^{(X)}$, $\sum_{i \in Z_1} k_i^{(X)}$ can be 0 or 1. To show (10), we consider these two cases separately:

Case 1. $\sum_{i \in Z_1} k_i^{(X)} = 0$:

We claim at least one of the following inequalities is true

$$\begin{aligned} Q_{2m} &\geq 2U/3 \times 1/2^m, \quad m = 0, \cdots, J-1 \\ Q_{2m+1} &\geq U/2 \times 1/2^m, \quad m = 1, \cdots, J-1 \end{aligned} \tag{11}$$

If the claim is not true, we reach a contradiction because

$$\begin{aligned} U &= \sum_{m=0}^{J-1} \sum_{i_0 \in Z_{2m}} k_{i_0}^{(X)} Q_{i_0}^{(X)} + \sum_{m=1}^{J-1} \sum_{i_1 \in Z_{2m+1}} k_{i_1}^{(X)} Q_{i_1}^{(X)} \overset{(a)}{<} \\ &\Big(\sum_{m=0}^{J-1} \sum_{i_0 \in Z_{2m}} k_{i_0}^{(X)} \frac{2}{3} \frac{1}{2^m} + \sum_{m=1}^{J-1} \sum_{i_1 \in Z_{2m+1}} k_{i_1}^{(X)} \frac{1}{2} \frac{1}{2^m} \Big) U \overset{(b)}{<} \\ &\Big(\sum_{m=0}^{J-1} \sum_{i_0 \in Z_{2m}} k_{i_0}^{(X)} \xi_{i_0} + \sum_{m=1}^{J-1} \sum_{i_1 \in Z_{2m+1}} k_{i_1}^{(X)} \xi_{i_1} \Big) U \overset{(c)}{\leq} 1 \times U, \end{aligned}$$

where (a) is due to the assumption that none of inequalities in (11) hold and using the fact that $Q_i^{(X)} \leq Q_j$ if $i \in Z_j$, (b) is due to the fact $\xi_i > \inf I_j$ if $i \in Z_j$, and (c) is due to the server's capacity constraint for feasible configuration $\mathbf{k}^{(X)}$.

Hence, one of the inequalities in (11) must be true. If $Q_{2m} \geq 2U/3 \times 1/2^m$ for some $m = 0, \cdots, J-1$, then (10) is true for configuration $\mathbf{k} = 2^m \mathbf{e}_{2m}$, while if $Q_{2m+1} \geq U/2 \times 1/2^m$ for some $m = 1, \cdots, J-1$, then (10) is true for configuration $\mathbf{k} = 3 \cdot 2^{m-1} \mathbf{e}_{2m+1}$.

Case 2. $\sum_{i \in Z_1} k_i^{(X)} = 1$:

In this case $\sum_{i \in Z_0} k_i^{(X)} = 0$. We further distinguish three cases for Q_1 compared to U: $Q_1 \geq \frac{2U}{3}$, $\frac{2U}{3} > Q_1 \geq \frac{U}{2}$, and $\frac{U}{2} > Q_1$. In the second case, we further consider two subcases depending on $\sum_{i \in Z_2} k_i^{(X)}$ being 0 or 1. Here we present the analysis of the case $\frac{2U}{3} > Q_1 \geq \frac{U}{2}$, $\sum_{i \in Z_2} k_i^{(X)} = 0$. The rest of the cases are either trivial or follow a similar argument, thus omitted to save space.

Let $U' := U - Q_1$, then one of the following inequalities has to be true

$$\begin{aligned} Q_{2m} &\geq U'/(3 \cdot 2^{m-2}), \ m = 2, \cdots J-1 \\ Q_{2m+1} &\geq U'/(3 \cdot 2^{m-1}), \ m = 1, \cdots J-1, \end{aligned} \tag{12}$$

otherwise, we reach a contradiction, similar to Case 1, i.e.,

$$\begin{aligned} U' &= \sum_{m=2}^{J-1} \sum_{i_0 \in Z_{2m}} k_{i_0}^{(X)} Q_{i_0}^{(X)} + \sum_{m=1}^{J-1} \sum_{i_1 \in Z_{2m+1}} k_{i_1}^{(X)} Q_{i_1}^{(X)} \overset{(a)}{<} \\ &2U' \Big(\sum_{m=2}^{J-1} \sum_{i_0 \in Z_{2m}} k_{i_0}^{(X)} \frac{2}{3} \frac{1}{2^m} + \sum_{m=1}^{J-1} \sum_{i_1 \in Z_{2m+1}} k_{i_1}^{(X)} \frac{1}{3} \frac{1}{2^m} \Big) \overset{(b)}{<} U' \end{aligned}$$

where (a) is due to the assumption that none of inequalities in (12) hold, and (b) is due to the constraint that the jobs in the configuration $\mathbf{k}^{(X)}$, other than the job types in Z_1, should fit in a space of at most $1/2$ (the rest is occupied by a job of size at least $1/2$). Depending on which of the inequalities in (12) is true, one of the configurations in $\mathcal{K}_{RED}^{(J)}$, with $k_1 = 1$, will satisfy (10). □

Using Proposition 1 and multi-step Lyapunov technique [31], we can prove the following theorem regarding the throughput of VQS. The proof details can be found in [30].

Theorem 3. VQS *achieves at least $\frac{2}{3}$ of the optimal workload $\rho^\star$, if arriving jobs have a minimum size of at least $1/2^J$.*

Hence, given a minimum job's resource requirement $u > 0$, J has to be chosen larger than $\log_2(1/u)$ in the VQS algorithm. Theorem 3 is not trivial as it implies that by scheduling under the configurations in $\mathcal{K}_{RED}^{(J)}$ (7), on average at most $1/3$ of each server's capacity will be underutilized because of capacity fragmentations, *irrespective* of the job size distribution F_R. Moreover, using $\mathcal{K}_{RED}^{(J)}$ reduces the search space from $O(\text{Exp}(J))$ configurations to only $4J-4$ configurations, while still guaranteeing $2/3$ of the optimal workload $\rho^\star$.

A natural and less dense partition could be to only consider the cuts at points $1/2^j$ for $j = 0, \cdots, J$. This creates a partition consisting of J sub-intervals $\widetilde{I}_j = I_{2j} \cup I_{2j+1}$. The convex hull of only the first J configurations of $\mathcal{K}_{RED}^{(J)}$ contains all feasible configurations of this partition. Using arguments similar to proof of Theorem 3, we can show that this partition can only achieve $1/2$ of the optimal workload $\rho^\star$. One might conjecture that by refining partition I (6) or using different partitions, we can achieve a fraction larger than $2/3$ of the optimal workload $\rho^\star$; however, if the partition is agnostic to the job size distribution F_R, refining the partition or using other partitions *does not* help. We state the result in the following Proposition.

Proposition 2. *Consider any partition X consisting of a finite number of disjoint sets X_j, $\cup_{j=1}^N X_j = (0, 1]$. Any scheduling algorithm that maps the sizes of jobs in X_j to $r_j = \sup X_j$*

(i.e., schedules based on upper-rounded VQ*s) cannot achieve more than* $2/3$ *of the optimal workload* $\rho^\star$ *for all* F_R*.*

Proof. The proof is based on constructing a counter example and can be found in [30]. □

Theorem 3 assumed that there is a minimum resource requirement of at least $1/2^J$. This assumption can be relaxed as stated in the following corollary.

Corollary 1. *Consider any general distribution of job sizes* F_R*. Given any* $\epsilon > 0$*, choose* J *to be the smallest integer such that* $F_R(1/2^J) < \epsilon$*, then the* VQS *algorithm achieves at least* $(1-\epsilon)\frac{2}{3}$ *of the optimal workload* $\rho^\star$*.*

Proof. Consider two systems in which jobs of size at most $1/2^J$: (1) are completely discarded from queue, or (2) join the queue but their resource requirement is resampled from F_R until it becomes greater than $1/2^J$. The two systems have the same job-size distribution, but their job arrival rate differs by a factor of $1-\epsilon$. These two systems can be used to find upper and lower bounds on the maximum throughput achieved by VQS. See our technical report [30] for details. □

Since the complexity of VQS algorithm is linear in J, it is worth increasing it if that improves maximum throughput. An implication of Corollary 1 is that this can be done adaptively as estimate of F_R becomes available.

VI. VQS-BF: Incorporating Best-Fit in VQS

While the VQS algorithm achieves in theory a larger fraction of the optimal workload than BF-J/S, it is quite inflexible compared to BF-J/S, as it can only schedule according to certain job configurations and the time until configuration changes may be long, hence might cause excessive queueing delay. We introduce a hybrid VQS-BF algorithm that achieves the same fraction of the optimal workload as VQS, but in practice has the flexibility of BF. The algorithm has two steps similar to VQS: Setting the active configuration is exactly the same as the first step in VQS, but it differs in the way that jobs are scheduled in the second step. Suppose the active configuration of server ℓ at time t is $\mathbf{k} \in \mathcal{K}_{RED}^{(J)}$, then:

(i) If $k_1 = 1$, the server will try to schedule the *largest-size job* from VQ_1 that can fit in it. This may not be possible because of jobs already in the server from previous time slots. Unlike VQS, when jobs from VQ_1 are scheduled, they reserve exactly the amount of resource that they require, and no amount of resource is reserved if no job from VQ_1 is scheduled.

(ii) Any configuration $\mathbf{k} \in \mathcal{K}_{RED}^{(J)}$ has at most one $k_j > 0$ other than k_1. Server attempts to schedule jobs from the corresponding VQ_j, starting from the *largest-size job* that can fit in it. Depending on prior jobs in server, this procedure will stop when either the number of jobs from VQ_j in the server is at least k_j, or VQ_j becomes empty, or no more jobs from VQ_j can fit in the server.

(iii) Server uses BF-S to possibly schedule more jobs in its remaining capacity from the remaining jobs in the queue.

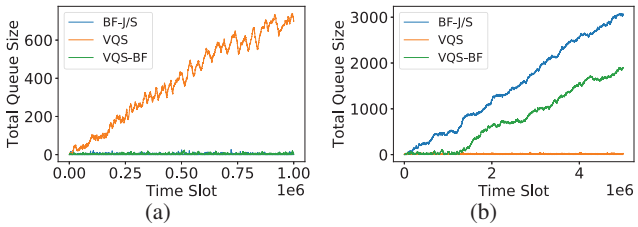


Fig. 3: (a) A setting where VQS is unstable, but BF variants are stable. (b) A setting where VQS is stable but BF variants are unstable.

The performance guarantee of VQS-BF is the same as that of VQS, as stated by the following theorem and corollary.

Theorem 4. *If jobs have a minimum size of at least* $1/2^J$*,* VQS-BF *achieves at least* $\frac{2}{3}\rho^\star$*. For a general job-size distribution* F_R*, if* J *is chosen such that* $F_R(1/2^J) < \epsilon$*, then* VQS-BF *achieves at least* $(1-\epsilon)\frac{2}{3}\rho^\star$*.*

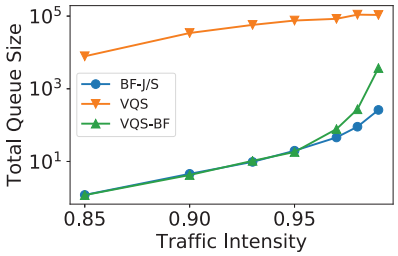
Proof. The proof is similar to that of Theorem 3. However, the difference is that the configuration of a server (jobs residing in a server) is not predictable, unless it empties, at which point we can ensure that it will schedule at least the jobs in the max weight configuration assigned to it, for a number of time slots proportional to the total queue length. The fact that the scheduling starts from the largest job in a virtual queue is important for this assertion, similarly to the importance of Best Fit in the proof of Theorem 2. See [30] for details. □

VII. Evaluation Results

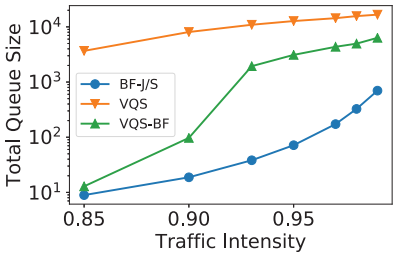
A. Synthetic Simulations

1) Instability of VQS *and tightness of* $2/3$ *bound.:* We first present an example that shows the tightness of the $2/3$ bound on the achievable throughput of VQS. Consider a single server where jobs have two discrete sizes 0.4 and 0.6. The jobs arrive according to a Poisson process with average rate 0.014 jobs per time slot and with each job size being equally likely. Each job completes its service after a geometric number of time slots with mean 100. Observe that by using configuration $(1,1)$ (i.e., 1 spot per job type) any arrival rate below 0.02 jobs per time slot is supportable. This is not the case though for VQS that schedules based on configurations $\mathcal{K}_{RED}^{(J)}$, so it can either schedule two jobs of size 0.4 or one job of size 0.6. This results in VQS to be unstable for any arrival rate greater than $2/3 \times 0.02 \approx 0.013$. Both of the other proposed algorithms, BF-J/S and VQS-BF, circumvent this problem. The evolution of the total queue size is depicted in Figure 3a

2) Instability of BF-*J/S:* We present an example that shows BF-J/S is not stable while VQS can stabilize the queues. Consider a single server of capacity 10 and that job sizes are sampled from two discrete values 2 and 5. The jobs arrive according to a Poisson process with average rate 0.0306 jobs per time slot, and job of size 2 are twice as likely to appear than jobs of size 5. Each job completes its service after a fixed number of 100 time slots . The evolution of the queue size is



(a) Job sizes$\sim$Unif $[0.01, 0.19]$



(b) Job sizes$\sim$Unif $[0.1, 0.9]$

Fig. 4: Comparison of the average queue size of different algorithms, for various traffic intensities, when job sizes are uniformly distributed in (a) $[0.01, 0.19]$ and (b) $[0.1, 0.9]$, in a system of 5 servers of capacity 1.

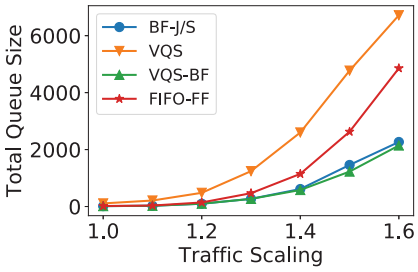


Fig. 5: Comparison of algorithms using Google trace for approximately 1,000,000 tasks. Traffic scaling varies from 1 to 1.6 and number of servers is fixed at 1000.

depicted in Figure 3b. This shows an example where VQS is stable, while both BF-J/S and VQS-BF are not.

To justify the behavior of the latter two algorithms, we notice that under both the server is likely to schedule according to the configuration $(2, 1)$ that uses two jobs of size 2 and one of size 5. Because of fixed service times, jobs that are scheduled at different time slots, will also depart at different time slots. Hence, it is possible that the scheduling algorithm will not allow the configuration $(2, 1)$ to change, unless one of the queues empties. However, there is a positive probability that the queues will never get empty since the expected arrival rate is more than the departure rate for both types. The arrival rate vector is $\boldsymbol{\lambda} = (0.0204, 0.0102)$ while the departure rate vector $\boldsymbol{\mu} = (0.02, 0.01)$.

VQS on the other hand will always schedule either five jobs of size 2 or two of size 5. The average departure rate in the first configuration is $\boldsymbol{\mu}_1 = (0.05, 0)$, and in the second configuration $\boldsymbol{\mu}_2 = (0, 0.02)$. The arrival vector is in convex hull of these two vectors as $\boldsymbol{\lambda} < 4/9\boldsymbol{\mu}_1 + 5/9\boldsymbol{\mu}_2$ and therefore is supportable.

3) Comparison using Uniform distributions: To better understand how the algorithms operate under a non-discrete distribution of job sizes, we test them using a uniform distribution. We choose $L = 5$ servers, each with capacity 1. We perform two experiments: the job sizes are distributed uniformly over $[0.01, 0.19]$ in the first experiment and uniformly over $[0.1, 0.9]$ in the second one. Hence $\bar{R}$ is 0.1 in the first experiment and 0.5 in the second one.

The service time of each job is geometrically distributed with mean $1/\mu = 100$ time slots so departure rate is $\mu = 0.01$. The job arrivals follow a Poisson process with rate $\mu L/\bar{R}\times$ jobs per time slot (and thus $\rho = \alpha L/\bar{R}$), where α is a constant which we refer to as "traffic intensity" and $L = 5$ is the number of servers in these experiments. A value of $\alpha = 1$ is a bound on what is theoretically supportable by any algorithm. In each experiment, we change the value of α in the interval $[0.85, 0.99]$. The results are depicted in Figure 4.

Overall we can see that VQS is worse than other two algorithms in terms of average queue size. Algorithms BF-J/S and VQS-BF look comparable in the first experiment for traffic intensities up to 0.95, otherwise BF-J/S has a clear advantage. An interpretation of results is that VQS and VQS-BF have particularly worse delays when the average job size is large, since large jobs cannot be scheduled most of the time, unless they are part of the active configuration of a server. That makes these algorithm less flexible compared to BF-J/S for scheduling such jobs.

B. Google Trace Simulations

We test the algorithms using a traffic trace from a Google cluster dataset [4]. We performed the following preprocessing on the dataset:

- We filtered the tasks and kept those that were completed without interruptions/errors.
- All tasks had two resources, CPU and memory. To convert them to a single resource, we used the maximum of the two requirements which were already normalized in $[0, 1]$ scale.
- The servers had two resources, CPU and memory, and change over time as they a updated or replaced. For simplicity, we consider a fixed number of servers, each with a single resource capacity normalized to 1.
- Trace events are in microsec accuracy. In our algorithms, we make scheduling decisions every 100 msec.
- We used a part of the trace corresponding to about a million task arrivals spanning over approximately 1.5 days.

We compare the algorithms proposed in this work and a baseline based on Hadoop's default FIFO scheduler [1]. While the original FIFO scheduler is slot-based [19], the FIFO scheduler considered here schedules jobs in a FIFO manner, by attempting to pack the first job in the queue to the first server that has sufficient capacity to accommodate the job. We refer to this scheme as FIFO-FF which should perform better than the slot-based FIFO, since it packs jobs in servers (using First-Fit) instead of using predetermined slots.

We scale the job arrival rate by multiplying the arrival times of tasks by a factor β. We refer to $1/\beta$ as "traffic scaling" because larger $1/\beta$ implies that more jobs arrive in a time unit. The number of servers was fixed to 1000, while traffic scaling varied from 1 to 1.6. The average queue sizes are depicted in Figure 5. As traffic scaling increases, BF-J/S and VQS-BF have a clear advantage over the other schemes, with VQS-BF also yielding a small improvement in the queue size compared

to BF-J/S. It is interesting that VQS-BF has a consistent advantage over BF-J/S at higher traffic, albeit small, although both algorithms are greedy in the way that they pack jobs in servers.

VIII. Discussion and Open Problems

In this work, we designed three scheduling algorithms for jobs whose sizes come from a general unknown distribution. Our algorithms achieved two goals: keeping the complexity low, and providing throughput guarantees for any distribution of job sizes, *without* actually knowing the prior distribution.

Our results, however, are lower bounds on the performance of the algorithms and simulation results show that the algorithms BF-J/S and VQS-BF may support workloads that go beyond their theoretical lower bounds. It remains as an open problem to tighten the lower bounds or construct upper bounds that approach the lower bounds.

We made some simplifying assumptions in our model but results indeed hold under more general models. One of the assumptions was that the servers are homogeneous. BF-J/S and our analysis can indeed be easily applied without this assumption. For VQS and VQS-BF, the scheduling can be also applied without changes when servers have resources that differ by a power of 2 which is a common case. As a different approach, we can maintain different sets of virtual queues, one set for each type of servers. Another assumption was that service durations follow geometric distribution. This assumption was made to simplify the proofs, as it justifies that a server will empty in a finite expected time by chance. Since this may not happen under general service time distributions (e.g. one may construct adversarial service durations that prevent server from becoming empty), in all our algorithms we can incorporate a stalling technique proposed in [11] that actively forces a server to become empty by preventing it from scheduling new jobs. The decision to stall a server is made whenever server operates in an "inefficient" configuration. For BF-J/S that condition is when the server is less than half full, while for VQS and VQS-BF, is when the weight of a configuration is far from the maximum weight over $\mathcal{K}_{RED}^{(J)}$.

Finally we based our scheduling decisions on a single resource. Depending on workload, this may cause different levels of fragmentation, but resource requirements will not be violated if resources of jobs are mapped to the maximum resource (e.g. as in our preprocessing on Google trace data). A more efficient approach is to extend BF-J/S to multi-resource setting, by considering a Best-Fit score as a linear combination of per-resource occupancies. We leave the theoretical study of scheduling jobs with multi-resource distribution as a future research.

References

[1] "Apache Hadoop," https://hadoop.apache.org, 2018.

[2] "Apache Spark," https://spark.apache.org, 2018.

[3] "Apache Hive," https://hive.apache.org, 2018.

[4] J. Wilkes, "Google Cluster Data," https://github.com/google/cluster-data, 2011.

[5] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[6] S. T. Maguluri, R. Srikant, and L. Ying, "Stochastic models of load balancing and scheduling in cloud computing clusters," in *Proceedings of IEEE INFOCOM*, 2012, pp. 702–710.

[7] A. L. Stolyar, "An infinite server system with general packing constraints," *Operations Research*, vol. 61, no. 5, pp. 1200–1217, 2013.

[8] S. T. Maguluri and R. Srikant, "Scheduling jobs with unknown duration in clouds," in *Proceedings 2013 IEEE INFOCOM*, 2013, pp. 1887–1895.

[9] ——, "Scheduling jobs with unknown duration in clouds," *IEEE/ACM Transactions on Networking*, vol. 22, no. 6, pp. 1938–1951, 2014.

[10] J. Ghaderi, "Randomized algorithms for scheduling VMs in the cloud," in *IEEE INFOCOM*, 2016, pp. 1–9.

[11] K. Psychas and J. Ghaderi, "On non-preemptive VM scheduling in the cloud," *Proc. ACM Meas. Anal. Comput. Syst. (ACM SIGMETRICS 2018)*, vol. 1, no. 2, pp. 35:1–35:29, Dec. 2017.

[12] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE Transactions on Automatic Control*, vol. 37, no. 12, pp. 1936–1948, 1992.

[13] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proc. of the ACM SIGOPS symposium on operating systems principles*, 2009, pp. 261–276.

[14] S. Tang, B.-S. Lee, and B. He, "Dynamic slot allocation technique for mapreduce clusters," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2013, pp. 1–8.

[15] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2015.

[16] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," *European Conference on Computer Systems - EuroSys*, pp. 1–17, 2015.

[17] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," *NSDI*, vol. 167, no. 1, pp. 24–24, 2011.

[18] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "Hug: Multi-resource fairness for correlated and elastic demands." in *NSDI*, 2016, pp. 407–424.

[19] M. Usama, M. Liu, and M. Chen, "Job schedulers for big data processing in hadoop environment: testing real-life schedulers using benchmark programs," *Digital Communications and Networks*, vol. 3, no. 4, pp. 260–273, 2017.

[20] "Hadoop: Fair Scheduler," https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html, 2018.

[21] "Hadoop: Capacity Scheduler," https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html, 2018.

[22] M. R Garey and D. S Johnson, "Computers and intractability: A guide to the theory of NP-completeness," *WH Freeman & Co.*, 1979.

[23] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 3, no. 4, pp. 299–325, 1974.

[24] C. Kenyon and M. Mitzenmacher, "Linear waste of best fit bin packing on skewed distributions," *Random Structures & Algorithms*, vol. 20, no. 3, pp. 441–464, 2002.

[25] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: A survey," in *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996, pp. 46–93.

[26] D. Shah and J. N. Tsitsiklis, "Bin packing with queues," *Journal of Applied Probability*, vol. 45, no. 4, pp. 922–939, 2008.

[27] E. Coffman and A. L. Stolyar, "Bandwidth packing," *Algorithmica*, vol. 29, no. 1-2, pp. 70–88, 2001.

[28] D. Gamarnik, "Stochastic bandwidth packing process: stability conditions via Lyapunov function technique," *Queueing systems*, vol. 48, no. 3-4, pp. 339–363, 2004.

[29] V. Nitu, A. Kocharyan, H. Yaya, A. Tchana, D. Hagimont, and H. Astsatryan, "Working set size estimation techniques in virtualized environments: One size does not fit all," *Proc. of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 1, p. 19, 2018.

[30] K. Psychas and J. Ghaderi, "Scheduling jobs with random resource requirements in computing clusters," https://goo.gl/CVz2Dc, Columbia University, Tech. Rep., 2019.

[31] S. Foss and T. Konstantopoulos, "An overview of some stochastic stability methods," *Journal of the Operations Research Society of Japan*, vol. 47, no. 4, pp. 275–303, 2004.