Progress-based Container Scheduling for Short-lived Applications in a Kubernetes Cluster

Yuqi Fu*, Shaolun Zhang*, Jose Terrero*, Ying Mao*, Guangya Liu†, Sheng Li‡, Dingwen Tao§
*Computer and Information Science, Fordham University, * Email: {yfu81, szhang253, jterrero1, ymao41}@fordham.edu
†IBM China Systems Lab, † Email: liugya@cn.ibm.com
‡Department of Computer Science, University of Georgia, ‡ Email: sheng.li@uga.edu
§Department of Computer Science, University of Alabama, § Email: tao@cs.ua.edu

Abstract*—In the past decade, we have envisioned enormous growth in the data generated by different sources, ranging from weather sensors and customer purchasing records to Internet of Things devices. Emerging data-driven technologies have been reforming our daily life for years, such as Amazon Personalize [1], which creates real-time individualized recommendations for customers according to multidimensional data analytics. It is, however, a challenging task to fully utilize and harness the potential of data, especially big data, due to Volume, Velocity, Variety, Variability and Value (5Vs) [2]. Most businesses thus choose to migrate their hardware demands to cloud providers, such as Amazon Web Service [3], which is powered by hundreds of thousands of servers. A cluster that builds up by a number of cloud servers is a basic management unit to provide shared computing resources. The typical structure of a cluster consists of managers and workers. When a job arrives at the cluster, as the first step, managers have to select a worker to host the incoming job. Traditionally, the selection process is based on the state of the workers, e.g., resource availability and specifications of jobs, e.g., labels, zones and regions. With respect to currently running jobs, we propose a progress based container placement scheme, named `ProCon`. When scheduling incoming containers, `ProCon` not only considers instant resource utilization on the workers but also takes into account the estimation of future resource usage. Through monitoring the progress of running jobs, `ProCon` balances the resource contentions across the cluster and reduces the completion time as well as the makespan. Specifically, extensive experiments prove that `ProCon` reduces completion time by up to 53.3% for a particular job and improves overall performance by 23.0%. Additionally, `ProCon` records an improvement of makespan for up to 37.4% when compared to the default scheduler available in Kubernetes.

***Index Terms*—Big Data, Deep Learning, Container, Docker, Kubernetes.**

I. INTRODUCTION

Enormous growth in data from various domains, such as commerce, education, military and government has dramatically transformed the way businesses make decisions, causing decision making processes to now be data-driven. As a traditional industry, agriculture [4] is one of the representative fields that has been reformed by data powered technologies. For example, Mothive [5], CropX [6] and Arable [7] are companies that design and deploy sensors to produce precise and radiometrically accurate plant-level vegetation indices with the aims of achieving smart farming management. Moreover, a drone-based data-driven harvest solution is offered for automated agronomy services to maximize efficiency, reduce waste, and improve the predictability and control of crops.

According to a report from Economist [8], "data is to this century what oil was to the last one: a driver of growth and change." However, it is very challenging to fully utilize and harness the potential data has to offer, especially when dealing with the characteristic 5Vs of big data [2], due to expensive infrastructure support, non-trivial technical requirement and high management cost. In smart farming industry, the above mentioned producing and harvesting of data are the very initial steps of the process. Considering the whole system, the collected raw data requires to be properly structured, stored and analyzed, which is challenging due to the aforementioned expensive infrastructure support, comprehensive technical requirements and high management cost. Consequently, cloud computing that provides elastic, on-demand and all-in-one services is naturally a top choice to take advantage of and empower raw data. Equipped with hundreds of thousands of a wide diversity of servers, Amazon Web Service [3], Microsoft Azure [9] and Google Cloud Platform [10] are the dominant players of the cloud service providers.

Through shared resources, e.g. CPU, memory, disk and network, affordable services, like data storage, computation, warehouse, and analytics, are offered to end customers. Containerization is a leading virtualization technology that enables the resource sharing and, at meanwhile, maintains isolation among users. With increasing workloads, cloud service providers use a cluster as the basic management unit. A cluster usually consists of a hierarchical structure of managers and workers, where each manager and each worker is either a virtualized container or a bare-metal server. Cluster management toolkits, such as Docker Swarm [11] and Kubernetes [12], are in duty for container orchestration.

Whenever a job arrives at the cluster, a manager needs to, as the first reaction, select a worker to host it. Selecting the most desirable worker in the cluster is a non-trivial task. Traditionally, the selection process is based on the states of the workers, e.g. resource availability and specifications from users, e.g. labels, zones and regions. For example, spread strategy [13] in Docker Swarm tries to distribute the containers evenly inside the cluster and balanced-resource allocation [14] in Kubernetes favors nodes with balanced resource usage rate. Moreover, in a Kubernetes cluster, administrators can specify a multi-layer

placement scheme that combines different priorities to make the worker selection process comprehensive.

The present commercial systems, however, fail to take existing jobs on the worker into account. Although the running jobs could reflect indirectly by the resource usage on a particular worker, the expected completion time, especially for short-living computing jobs, should be taken into consideration. Assuming there is a 3-node cluster that configures to 1 manager and 2 workers. And that currrently, 3 jobs are running in this cluster, with 2 of them are being run on worker-1 and the other one being hosted by worker-2. With spread strategy, the incoming 4th job will be assigned to worker-2 to maintain an even distribution. It works perfectly fine if the existing jobs would elapse for a while after the assignment. But, if the 2 jobs on worker-1 are expected to finish in a second, the previous assignment would downgrade system performance.

In this work, we propose a PROgress-based CONtainer placement scheme, named `ProCon`. At runtime, `ProCon` studies progress values, which is commonly available for batch processing (e.g. Hadoop 3.0 [15]) and machine learning (e.g. Tensorflow [16]) applications, to estimate the expected completion time. Combining resource usage with expected completion time, `ProCon` selects a node to balance the contention on the workers in the cluster.

The main contributions of this paper are summarized as follows.

- We introduce the concept of contention rate, a measurement of the magnitude of the degrees in resource completion across the cluster.
- We propose `ProCon` with a suite of algorithms that monitor the progress of running containers and estimate their expected completion time. Additionally, it utilizes the collected data to calculate the expected contention rate and assign the incoming containers to balance the resource contention on the workers.
- We implement `ProCon` on top of a dominant container orchestration tool, Kubernetes, and evaluate it with popular containerized deep learning application. Through intensive experiments, `ProCon` achieves significant improvement on completion time for a particular job, of up to 53.3%, an overall reduction of up to 23.0% and records a decrease of makespan for up to 37.4%.

II. Related Work

With prevalence of data-driven businesses, demands for various services in the cloud, such as storage [17], computing [18], analytics [19] and learning [20], have increased dramatically in the past decade.

In this domain, virtualization is one of the fundamental technologies that powers the backend side of cloud computing. As an emerging virtualization solution, containerization [21] is replacing the traditional virtual machine due to its platform independent, resource lightweight, and flexible deployment [22]. Container technology has been studied to improve the services in the cloud from different perspectives [23]–[30].

Despite the benefits of deploying containers, researchers in [23] show that the startup latency is considerably larger than expected, this is due to a layered file system and distributed image architecture, in which copying package data accounts for most of container startup time. The authors present Slacker to quickly provision container storage using backend clones and minimize startup latency by lazily fetching container data. Additionally, CoMICon [30] addresses the same problem by sharing the image in a cooperative manner and further reducing the provisioning time. Moreover, SCoPe [31] presents a statistical model to manage the provisioning time for large scale containers.

Focusing on wait time in a congested datacenter, Big-C [24] is proposed to minimize the queuing delays of short jobs while maximizing resource utilization. It includes immediate and graceful preemptions, and shows their effectiveness and tradeoffs with loosely-coupled MapReduce as well as iterative workloads. While Big-C uses available runtime estimates to perform task placements, Kairos [25] achieves low latency and high resource utilization without task runtime estimation. It introduces a distributed approximation of the Least Attained Service scheduling policy. Tagrgeting on a specific type of applications, Minos [26] further reduces the tail latency of in-memory key-value stores. It implements size-aware sharding, a new technique that assigns small and large requests to a disjoint set of cores.

When deploying services in a production environment, cloud service providers utilize a cluster of physical machines to host them. The toolkits for container orchestration are, usually, required for managing the containers in a cluster. Docker Swarmkit [32] and Kubernetes [12] are dominant cluster management tools in the market. As the first step in initialization, authors of [33] propose Draps to place the containers based on different patterns of their resource demands. While Draps provides a general solution for container placement, Stratus [34] and PIVOT [35] investigate new cluster schedulers that are specialized for orchestrating big data applications on virtual clusters. Besides, FlowCon [36] is proposed to optimize the performance of containerized deep learning applications.

While existing researches optimize the system from various points of views, few of them take the current running jobs and their expected completion time into account. In this paper, we propose a new scheduler, named `ProCon`, which places container based not only on resource utilization at present but also on the estimate of future resource usage in order to minimize the contention rate in the cluster.

The remainder of this paper is organized as follows. In Section III, we introduce the background with motivating examples of this project. In Section IV and Section V, we present the `ProCon` architecture and algorithms in details. We carry out the extensive evaluation of `ProCon` in the cloud (in Section VI) and conclude the paper in Section VII.

III. Background and Motivation

In this section, we briefly introduce containerized applications and motivate our work with an example.

A. Container Orchestration

Containerization is a virtualization technique that enables applications to encapsulate all necessary dependencies into a sandbox in order to build a platform-independent runtime environment. Nowadays, many public cloud providers offer Containers as a Service (CaaS) to simplify deployment of containerized applications in the cloud. Generally, there are two types of containers in the cloud.

- Long-lived containers: They aim to provide targeted services that would last for a long time, such as web, logging and storage services.
- Short-lived containers: They mainly focus on computing-oriented services, which will expire when finished. The representative services are batch processing for Hadoop Yarn, Spark and iterative processing for Tensorflow, Pytroch (deep learning platforms).

From the cloud service provider's perspective, running containerized applications creates an abstraction layer that handles cluster management. The typical structure of a cluster of containers consists of manager and worker nodes. Worker nodes are responsible for running containers with workloads that are submitted by the users; on the other hand, manager nodes accept specifications and are responsible for reconciling desired states with actual cluster states. Docker [11] and Kubernetes [12] are leading open-source container orchestration toolkits. A container can be initiated with specifications from users, e.g. resource limits, zones, and labels, etc. In the system, a scheduler that resides in the manager is responsible for various scheduling decisions, which are made taking into account user specified requirements, cluster configurations as well as current states. Whenever there is an incoming request, managers have to, first, select a worker to host the container. The node selection is a two-step process.

1) The scheduler filters out workers that fail to meet requirements, e.g. `disktype: ssd` in the YAML file of a Kubernetes container indicates that the host must have a solid state disk.
2) The scheduler ranks the remaining candidates based on default or a user specified placement scheme. This scheme usually combines multiple factors, such as node state, resource availability and current workload to prioritize workers.

B. Motivation of ProCon

`ProCon` investigates the second step of the container placement process. In particular, `ProCon` focuses on the short-lived containers, which provide popular services like execution of data mining algorithms, batch processing for big data and training neural network models.

While the approaches in literature take many factors into account, they fail in properly addressing the characteristics of short-lived containers that release used resources when jobs terminate. Fig. 1 shows a simplified illustrative example. Suppose there are 3 containers in the cluster, one on Worker-1 and two on Worker-2. The numbers under the boxes represent

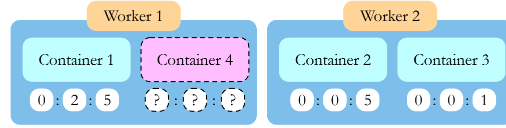


Fig. 1: Illustrative Example

the remaining time, in the format of hour, minute and second, for each of them. Without the remaining time, the manager should obviously place the incoming 4th container to worker-1 in order to balance the workload. However, if the scheduler is well aware of the expected completion time for the running containers, the incoming 4th one should be assigned to worker-2 since running jobs on worker-2 will finish in 5 seconds.

The illustrative example is based on estimating the completion time of existing containers. The actual time cost depends on mainly three factors.

- Job itself: The tasks that are executed inside a container are determined by the users. The completion time varies with algorithms, implementation and data sets, which are out of the control of the computing system.
- Resource availability: The tasks utilize different resources, such as CPU and network, to keep progressing. Usually, the maximum resource usage allowed is a fixed number based on the parameters specified by the users and the configuration of the workers.
- Workload on workers: The containers on the same worker compete for resources. The more resources available to this container , the faster it moves.

Despite limited information about job itself, the progress of short-lived containers can be estimated given the real-time resource usage, which we can obtain as a system administrator. Fig. 2 shows the time cost for 3 jobs, MapReduce based wordcount (shown as Hadoop-Map and Hadoop-Reduce), Spark based KMeans and Tensorflow based LeNet. For the MapReduce job, we record the cost to increase one percentual point of the total progress (excluding the shuffling phase) and for Spark and Tensorflow jobs we record the cost associated with each iteration. The figure illustrates that the cost is stable and floats within a small cell. While the jobs are running in alone without resource completion, a stable cost motivates us to estimate the completion time with the help of resource usage.

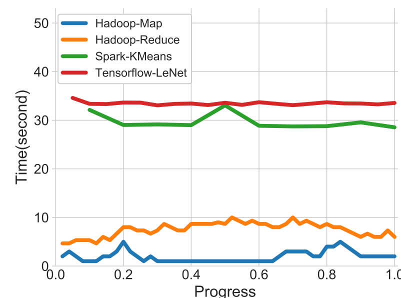


Fig. 2: Progress of representative workloads

IV. System Design

ProCon builds on top of Kubernetes, which is a popular container-orchestration system for automating application deployment, scaling, and management. In this section, we present the system design of ProCon in detail, including its design logic and functionalities of key modules.

A. Framework of Kubernetes

A Kubernetes cluster consists of workers and managers. A pod in Kubernetes is group of containers that are deployed together on the same host. In the cluster, each pod consists of one or more containers and each worker can host one or many pods.

There are 6 basic components in Kubernetes. The manager nodes consist of **API Server**, **Controller Manager** and **Scheduler** and **etcd**. The **Kuberlet** and **Service Proxy** resident in workers.

- API Server: It is the main management point of the entire cluster and processes REST operations, validates them, and updates the corresponding objects in storage.
- Controller Manager: It runs controllers, which are the background threads that handle routine tasks.
- Scheduler: It watches for newly created Pods that have no node assigned and is responsible for placement of pods on workers.
- etcd: It is a distributed data storage solution that stores all the data, e.g. configuration, state, and metadata.
- Kuberlet: It is responsible for maintaining a set of pods, which are composed of one or more containers, on a local system.
- Service Proxy: It maintains network rules on nodes, e.g. implementing a form of virtual IP for services.

B. ProCon Modules

As demonstrated in Fig. 3, ProCon consists of three modules, a ProCon Scheduler and a Log Analyst and on the manager side, and a Container Monitor on the worker side. Each module runs independently and exchanges information about the jobs inside the containers as well as worker status. Their functionality is shown in detail as below.

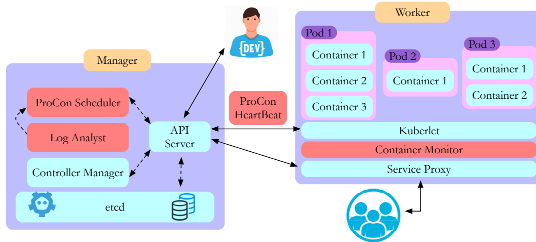


Fig. 3: ProCon System Architecture

1) Container Monitor: Runs on the worker nodes, the container monitor keeps track of the status of running jobs. For example, it maintains the number of completed iterations of a Tensorflow training job along with its time cost for each iteration. During the time period, the average resource consumption of this container will also be recorded. These records will be stores as log files in a persistent volume.

2) Log Analyst: The log analyst runs on the manager node, who has a global view of all workers. It analyzes the log files generated by the container monitors in workers. Based on these log files, it calculates the required parameters for the scheduling algorithm, such as contention rate (details in section V), and communicates with the scheduler for the final decision on the container placement.

3) ProCon Scheduler: ProCon scheduler is a key module of the system. It gathers the information from the log analyst and calculates a score for each of the worker nodes. ProCon scheduler selects the worker to host an incoming container according to the scores. Besides the ranking of workers, it can apply additional priority algorithms or policies to accommodate the system needs.

V. ProCon Progress-based Container Placement

ProCon aims to improve the system performance by placing pending containers to the most desirable worker in the cluster. To achieve this goal, ProCon not only considers the current states of the workers but also future states, related with the expected completion time of running containers on them. In this section, we present how ProCon estimates the finish time for each container, calculates contention rate and selects the workers.

A. Completion Time Estimation

As discussed in Section III, how fast the computing job progresses depends on the job itself, which includes algorithms, implementation , datasets, etc. In addition, it also affects the amount of resources allocated to it. While the system has no clues regarding the first factor without scanning files inside a container, it is a constant factor when this particular job is running. For example, if users want to train a deep learning model with recurrent neural networks, they have to, first, design algorithms, implement them, and then specify datasets for their neural networks, and finally, package everything into a container and start the training process in the cloud. Once the container is running, the job itself will not change anymore. This constant factor can be indirectly reflected by the progress rate. Given the same resource amount, a well-designed algorithm should be less time-consuming and the same algorithm with a smaller dataset should cost less time to finish.

With the analysis above, finding the progress rate is crucial for the estimation ofthe completion time. Since ProCon focuses on short-lived computing applications, such as batch processing and deep learning, we define the progress rate as the percentage of the maximum number of iterations allowed that have already been computed. Or, the percentage of the amount of the total data that is to be processed that has been processed already. For example, if we need a maximum of 10K iterations to train a deep learning model and currently iteration number 1000 is being executed, the progress rate is 10%; if a data processing job, e.g. Hadoop WordCount, is expected

to process 5GB of data, and 2GBs of data have already been processed, then the progress rate is 40%.

In ProCon, the applications report their progress rate periodically, e.g. every time an iteration is completed, or every time the total progress increases by at least one percentual point, we consider multiple jobs running concurrently in a cluster with multiple workers. We use J_{id} to represent the job ID and W_i as the worker ID. Since the report of progress rate is a per-job record, we utilize Equation 1 to calculate the time cost for the latest progress report at given time t.

$$f(J_{id}, t) = P(J_{id}, n) - P(J_{id}, n-1) \tag{1}$$

, where $P(J_{id}, n)$ is the timestamp when J_{id} submits its nth (latest) report and $f(J_{id}, t)$ represents the cost of the latest two reports at time t.

Besides the job itself, the expected completion time also depends on the resources that is available to it. We can obtain the resource usage for each container by using the various tools. We define function $R(J_{id}, t)$, that returns the amount of resources being occupied by J_{id} at time t. Because the resources are shared by all running jobs on the same worker, we can calculate the relative resource usage though Equation 2

$$u(J_{id}, t) = \frac{R(J_{id}, t)}{\sum_{J_{id} \in W_i} R(J_{id}, t)} \tag{2}$$

Combining both equations, the remaining time indicator (i) for J_{id} can be computed through Equation 3.

$$i(J_{id}, t) = \frac{f(J_{id}, t)}{u(J_{id}, t)} \times (MAX - P(J_{id}, n)) \tag{3}$$

, where MAX is a constant value that represents the maximum progress that the job can achieve, e.g. 100 percents or user-defined maximum number of iterations.

TABLE I: Notation Table

Notation	Description
$J_{id} \in J$	The job ID in the cluster
$W_i \in W$	The worker ID in the worker set
$R(J_{id}, t)$	The resource usage for J_{id} at time t
$u(J_{id}, t)$	The relative resource utilization rate of J_{id} at time t
$L(W_i)$	The resource limit on the worker
$f(J_{id}, t)$	The time difference of the last two reports of J_{id} at time t
$P(J_{id}, n)$	The nth progress rate report of J_{id}
$i(J_{id}, t)$	The remaining time indicator for J_{id} at time t
$RC(W_i, t)$	The number of running container on the W_i at time t
$erc(W_i, t)$	The expected number of running container on W_i at time t
S	The candidate set
MAX	The maximum value of progress that defined by the user
$c(W_i)$	The contention rate of W_i

In Table I, we summarize the parameters and functions that are used for analysis and algorithms. Please note that function and parameter names start with upper case letters are either constant or can be obtained directly through existing tools or APIs. The function and parameter names begin with lower case letters can be calculated from others.

Algorithm 1 Contention Rate of W_i

```
1:  System Initialization: J_id, W_i, L(W_i)
2:  Parameters: P(J_id, n), RC(W_i, t), R(J_id, t), MAX

3:  if Σ_{J_id ∈ W_i} R(J_id, t) < ×L(W_i) then
4:      W_i.upper_limit = False

5:  for J_id ∈ W_i do
6:      u(J_id, t) = R(J_id, t) / Σ_{J_id ∈ W_i} R(J_id, t)
7:      i(J_id, t) = f(J_id, t) / u_{J_id} × (MAX − P(J_id, n))

8:  for ∀J_id ∈ W_i do
9:      Find Max(i(J_id, t))

10: for t = Sys(t); t < Max(i(J_id, t)), t++ do
11:     for J_id ∈ W_i do
12:         if i(J_id, t) > t then
13:             erc(W_i, t).insert(J_id)

14: t = Sys(t)
15: while t < Max(i(J_id, t)) do
16:     c(W_i) = c(W_i) + 1 × (|erc(W_i, t)|−1)
17:     t++
```

B. Calculate Contention on the Workers

Since the number of running containers on each worker significantly influences the completion time of jobs, we denote $RC(W_i, t)$ to be the function that returns the number of existing containers, which is directly available to managers. The larger the number of containers running concurrently, the more intense the contention for resources. Given the function $RC(W_i, t)$, the contention rate of W_i can be represented by Equation 4. To simplify the integral of Equation 4 in practice, we convert it to a summation problem using one-second time intervals.

$$c(w_i) = \int_{t=0}^{\infty} RC(W_i, t) \mathrm{d}t \tag{4}$$

The system needs to calculate the contention rate based on the existing containers and their expected completion time. In ProCon, Algorithm 1 runs on each worker node. As shown on Line 1 and 2, each worker maintains required data such that $J_{id}, W_i, L(W_i)$ are the information from the cluster itself, $RC(W_i, t), R(J_{id}, t)$ are available from APIs, and $P(J_{id}, n), MAX$ can be obtained from progress reports.

When the algorithm starts on W_i, it first checks the overall resource utilization. In a scenario that the utilization rate is lower than resource limits, ProCon marks this worker's "upper_limit" flag to false, which will be used later in the node selection process. When the flag is false, it means that no matter how many jobs a running on it, they cannot fully utilize the resources (Line 3 - 4).

Next, for each J_{id} on W_i, Algorithm 1 calculates the relative resource utilization rate, $u(J_{id}, t)$. Based on $u(J_{id}, t)$, it, then,

computes the remaining time indicator, $i(J_{id}, t)$ for J_{id} at time t (Line 5 - 7). At this stage, every J_{id} that runs on W_i has an indicator that suggests the expected completion time. The algorithm finds the largest $i(J_{id}, t)$ value, which reflects the time point when all the current running containers will be released due to the completion of the jobs (Line 8 - 9).

The function, $RC(W_i, t)$, returns the number of running containers at time t and can be directly obtained from APIs. The APIs, however, cannot provide any information beyond the current time. Therefore, starting from the current system time to Max$i(J_{id}, t)$, the algorithm calculates the expected running containers and formulates it into a function, $erc(W_i, t)$, which returns an expected value of $RC(W_i, t)$ (Lin 10 - 13).

Finally, Algorithm 1 resets time t to the current system time (Line 14) and calculates the contention rate for W_i based on the $erc(W_i, t)$ (Line 16 - 17). Different from the value of the remaining time, $i(J_{id}, t)$ is an indicator of it. Thus, different from Equation 4, the algorithm uses 1 for precision.

C. ProCon Container Placement

The Container Monitor on each worker runs Algorithm 1 to calculate the contention rate. It writes the results to a log file that is stored in a persistent volume, which can be accessed from the manager side. The Log Analyst, which resides on the manager, reads the files in the persistent volume and prepares the necessary per-job parameters for Algorithm 2.

Whenever an incoming job arrives at the cluster, it triggers the manager to start `ProCon` Scheduler, who is going to execute Algorithm 2. `ProCon`Scheduler first initializes the system, e.g. worker set W, job set J and candidate set S, as well as the required parameters that prepared by Log Analyst (Line 1 - 2).

After initialization, it enumerates the all the workers to find out the one with a false "upper_limit" flag. These workers will be added to the candidate set S (Line 3 - 5). For these workers, no matter how large their contention rates are, they will be considered as a host for the incoming job. The logic behind it is that due to design and implementation, some jobs cannot consume all the resources on workers. This feature fails to reflect by the relative resource utilization rate, $r(J_{id}, t)$. In this case, excluding a worker solely based contention rate will result in imbalanced workload. If S is not empty, Algorithm 2 ranks the candidates with their $c(W_i)$ and returns the one with minimum value of contention rate (Line 6 - 9).

If S is empty, all the workers will be considered as the candidates. For each J_{id} on W_i, `ProCon` calculates $f(J_{id}, t)$ based on progress reports (Line 10 - 14). Additionally, it calculates $u(J_{id}, t)$ under the hypothesis that the incoming job has been added to W_i and presumes the resource will be evenly distributed with portion of $\frac{1}{RC(W_i,t()+1}$ afterwards (Line 15).

Based on new values of $f(J_{id}, t)$ and $u(J_{id}, t)$, Algorithm 2 updates the remaining time indicator, $i(J_{id}), t$, expected running containers, $erc(J_{id}, t)$ and contention rate, $c(W_i)$, where the new value suggests the degree of resource contentions if the system place the incoming job on a particular worker W_i (Line 16 - 19).

Algorithm 2 Container Placement on the Manager

```
1:  System: W, J, S = {}
2:  Parameters: P(J_id, n), RC(W_i, t), R(J_id, t), MAX

3:  for W_i ∈ W do
4:      if W_i.upper_limit = False then
5:          S.inset(W_i)

6:  if |S| > 0 then
7:      for W_i ∈ S do
8:          Find Min(c(W_i))
9:      Return W_i with minimum c(W_i)

10: else
11:     for W_i ∈ W do
12:         S.inset(W_i)
13:         for J_id ∈ W_i do
14:             f(J_id, t) = P(J_id, n) − P(J_id, n − 1)
15:             u(J_id, t) = 1 / (RC(W_i, t() + 1)
16:             Re-calculate i(J_id, t)
17:             t = Max(i(J_id, t))
18:             Re-calculate erc(W_i, t)
19:             Re-calculate c(W_i)
20:         Find Min(c(W_i))
21:     Return W_i with minimum c(W_i)
```

Finally, `ProCon` finds the W_i with the minimum value of $c(W_i)$ and select it as the host of the new container.

VI. Evaluation

In this section, we evaluate the performance of `ProCon` through intensive cloud-executed experiments.

A. Experimental Framework

We integrate `ProCon` into Kubernetes 1.15 and implement it as plug-in modules that reside on both manager and worker nodes. It receives tasks from the manager, and then directs the given tasks to a selected worker for execution. We build the testbed on NSF Cloudlab [37], which is hosted at the University of Utah, Downtown Data Center. Specifically, we use multiple M510 as our physical machines that contain two 8-core Intel Xeon D-1548 at 2.0 GHz, 64GB ECC Memory, and 256 GB NVMe flash storage. To evaluate the system, we build two clusters,

- **Cluster 1**: 1 Manager and 4 Workers.
- **Cluster 2**: 1 Manager and 7 Workers.

`ProCon` focuses on the short-lived containers. As a representative and extremely-popular type of computing workload, we evaluate `ProCon` with various deep learning applications through two widely-used platforms, Tensorflow [16] and Pytorch [38]. Table II lists the workloads used in the experiments.

B. Experiment Setup and Evaluation Metrics

In `ProCon`, Algorithm 1 runs on each worker to prepare the necessary information required by Algorithm 2. The objective

TABLE II: Tested Deep Learning Models

Model	MAX	Plat.
Variational Autoencoders (VAE) [39]	15	P/T
Modified-NIST (MNIST)-CNN [40]	200	P/T
Deep Regression Neural Network [41]	5000	P
Bidirectional-RNN [42]	10000	T
LeNet-CNN [43]	5	T

of `ProCon` is to place the container in the most appropriate worker node in order to reduce the overall completion time for jobs in this cluster. Since deep learning applications are computation-intensive , they are more sensitive to CPU than memory spaces and network bandwidth. The following three metrics are considered in our experiments.

- Completion Time: the completion time of each individual job in the cluster.
- Overall System Performance: the total length of the schedule for all the jobs in the system (makespan) and the average completion time.
- Contention Rate: the contention rate on each worker, which has a significantly impact on the other two factors.

To ensure a comprehensive evaluation, we design the following submission schedules, **Fixed Schedule**: the time to launch a job follows a fixed interval. It simulates an administrator controlled cloud environment. **Random Schedule**: the time to launch a job is randomly selected within an interval. It simulates an user-specified cloud environment.

In addition, we compare `ProCon` with the default scheduler in Kubernetes. In the rest of the evaluation part, we denote DS to be the default scheduler.

C. Fixed schedule with one job type

We fix the schedule with an interval of 30 seconds such that the jobs are submitted at time 0, 30, 60, ..., etc. Additionally, the type of jobs is fixed in this experiment. We submit 20 jobs of Bidirectional-RNN on Tensorflow to the cluster to evaluate `ProCon` on **Cluster 1**.

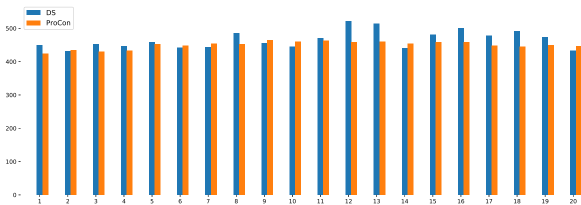


Fig. 4: Fixed schedule with 20 Bidirectional-RNN jobs

Completion Time and Makespan: Fig.4 presents the completion time of jobs in this experiment. Since there is only one type of jobs in the cluster, as we can see that the values for each of them is stable. There are 13 out of 20 jobs record a reduction on the completion time. The average improvement of the 13 jobs is 7.4% (448.7s v.s. 484.6s). The 7 jobs, which have a longer execution time, get a 2.2% increase (451.9s v.s 441.7s). As for makespan, which reflects the overall system performance, while reducing the average completion time, `ProCon` remains stable when comparing with DS (1005.6s v.s. 1017.1s).

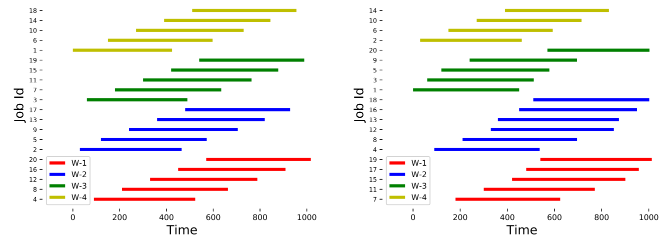


Fig. 5: `ProCon` with fixed schedule and 20 jobs

Fig. 6: DS with fixed schedule and 20 jobs

Contention Rate: Fig. 5 and Fig. 6 illustrate details of the container placement for `ProCon` and the DS. First of all, we find that the distribution with DS is imbalance since there are 6 jobs on Worker-1 and 4 jobs on Worker-2. This is because DS in Kubernetes prioritizes the worker based on instant resource usage when the incoming job arrives. However, there is a delay when fetching the resource usage due to the latency when executing jobs. Considering the fixed schedule and one job type, `ProCon` acts like a round-bin algorithm such that each worker is assigned 1 job and then rotate. When the the 5_{th} job arrives at the cluster, each of the workers has one running container and, of course, the worker, who hosts the first job, is expected to finish sooner than other. Therefore, `ProCon` assigns Job-5 to Worker-4.

When calculating the contention rate, `ProCon` outperforms DS on Worker-1 (1323 v.s. 1898), Worker-3 (1325 v.s. 1370) and Worker-4 (1275 v.s. 1331), but fails on Worker-2 (1327 v.s. 970), which has less number of jobs on it. Please note that `ProCon` aims on balancing the resource contention across nodes and it does not focus on minimizing the value on a particular worker. The standard deviation values are 25 and 382, which suggests `ProCon` is more stable across different workers.

Remarks: `ProCon`'s achievement is limited (overall 7.4%) under a fix submission pattern. The reason lies in the fact that, with a fixed interval of 30s and fixed job type to Bidirectional-RNN on Tensorflow, both resource usages and demands are very stable in the system, which reduce rooms for optimization.

D. Random schedule with multiple types of jobs

In this experiment, we use the same testbed (**Cluster 1**), but randomly generate a submission schedule within the interval 0 to 600s for 20 jobs, which randomly selected from Table II.

Completion Time and Makespan: Fig. 7 shows a different trend such that the completion time varies due to different job types as well as random submissions. With this schedule, 14 out of 20 jobs get improved on completion time. There is a 26.1% average reduction among those jobs and the largest gain is found on Job-18, which reduces 53.3% from 406.1s to 189.8s. To achieve the significant improvement, however, we found that the completion time of Job-9 increases significantly

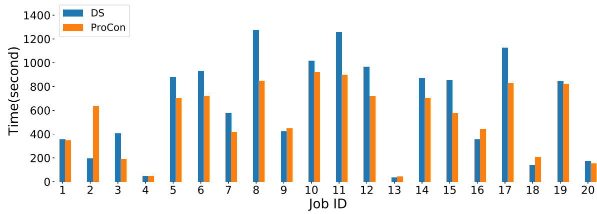


Fig. 7: Random schedule [0, 600s] with 20 random jobs

from 195.4s to 640.0s. This is because Job-9 is running on Worker-4 with DS and it is the only job on Worker-4 for the majority of its execution time. Besides, there are 5 other jobs record an increase on completion time, Job-2, Job-9, Job-13, Job-16, Job-18. The loss in these jobs is due to the imbalance workload distribution on DS, where some workers, at certain point, has much less jobs than others. The container assignment on DS benefits Job-2 significantly, and limited number of other jobs, however, it sacrifices many others. In contrast, Job-2 is hosted by Worker-2 in ProCon and during most of its lifetime, there are two other jobs running concurrently(demonstrated on Fig. 8 and Fig. 9).

Overall, ProCon achieves 23.0% reduction on the average completion time among all 20 jobs. Furthermore, ProCon records a 15.3% decrease on the makespan of this schedule, from 1611.2s to 1364.8s.

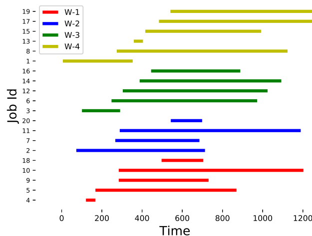


Fig. 8: ProCon with random schedule and 20 jobs

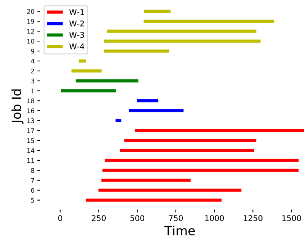


Fig. 9: DS with random schedule and 20 jobs

Contention Rate: The improvement of ProCon is obtained through efficient container placement. Fig. 8 and Fig. 9 present the detailed container placement of ProCon and DS, respectively. Comparing the figures, DS clearly leads to an imbalanced placement. For example, Worker-2 is idle until time 359s, when Job-13 submits to the cluster. In ProCon, the containers are distributed more appropriately. With the expected completion time in mind, the second container that hosts by Worker-1, Worker-3 and Worker-4 is started at time 167s, 248s, 274s, and the first one on the respective worker finishes at time 169s, 290s, and 353s. As we can see from the Fig. 8, the overlaps of first and second jobs on Worker-1, 3 and 4 are small, which result in less resource contention during the period. On Worker-2, however, the overlap of the first two jobs is longer than others. This is due to the fact that before the first one completes, a third job is assigned to this worker that increased the level of contention and the execution time for all of them becomes longer.

The contention rate of 4 worker nodes for ProCon and DS are 1239, 6323 (Worker-1), 997, 91 (Worker-2), 1791, 250 (Worker-3) and 2112, 2344 (Worker-4). The standard deviation of contention rates are 508 and 2901 for ProCon and DS. The lower value indicates that ProCon can balance the resource contentions.

E. Scalability of ProCon

Next, we evaluate our ProCon with an increased number of jobs and a more workers in the cluster.

1) Lager workload: We conduct the experiments on **Cluster 1**, which is the same one as we used in the previous experiments. We submit 30 and 40 randomly selected jobs within the interval [0, 600] in these experiments.

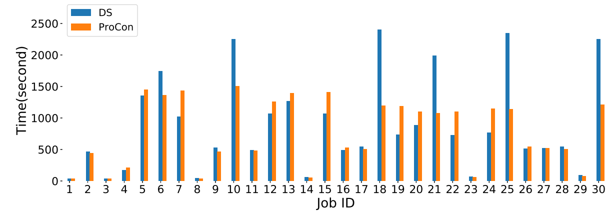


Fig. 10: Random schedule [0, 600s] and 30 random jobs

Completion Time and Makespan: Fig. 10 and Fig. 11 present the results from the experiments. As before, similar trends can be found on the figures such that ProCon outperforms DS generally. With 30 jobs running in the cluster, ProCon improves 18 of them, with an average reduction of 34.8% and 11.3% overall improvement among all 30 jobs. Moreover, ProCon lowers completion time for 26 jobs when 40 of them are submitted in the system. The average improvement of those 26 jobs is 22.4% and overall reduction is 10.2%. As we can see from the results, the improvement of overall average completion time decreases when the cluster has, not only a large amount of jobs, but also a significantly denser schedule. If the interval of two consecutive jobs is too small, it is difficult for ProCon to obtain necessary information and select the most desirable node. When considering the makespan of the system, however, ProCon still wins over DS by a large margin, 36.6% (1793.3s v.s. 2828.1s) and 28.2% (1908.2s v.s. 2656.2s) for 30-job and 40-job experiment, respectively.

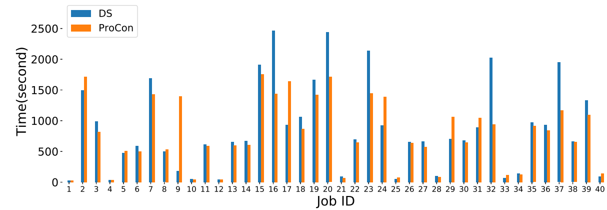
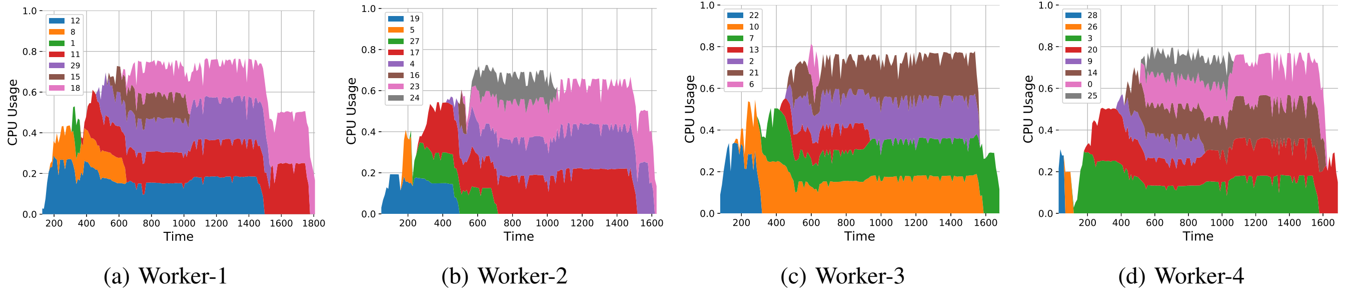


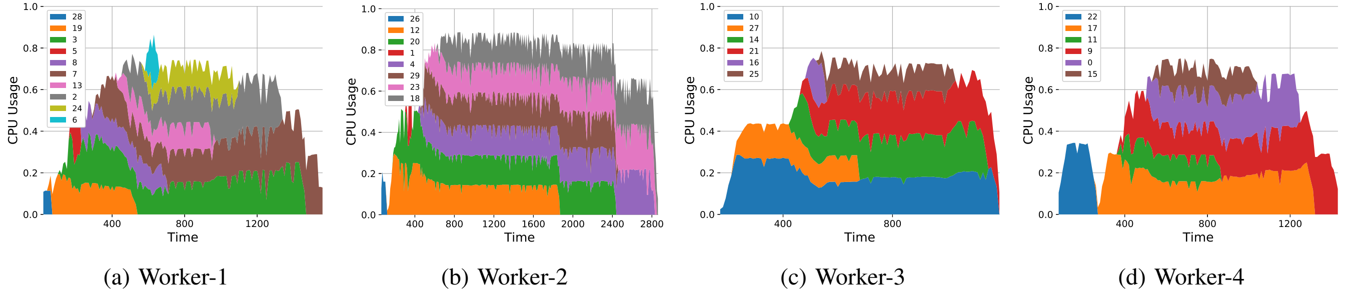
Fig. 11: Random schedule [0, 600s] and 40 random jobs

Contention Rate: The contention rate for each worker in 30-job experiment is 4188 v.s. 4226, 3527 v.s. 10275, 4220



(a) Worker-1 (b) Worker-2 (c) Worker-3 (d) Worker-4

Fig. 12: CPU usage for 30-job experiment with ProCon



(a) Worker-1 (b) Worker-2 (c) Worker-3 (d) Worker-4

Fig. 13: CPU usage for 30-job experiment with DS

v.s. 2525, 4765 v.s. 2378 for ProConand DS. Obviously, ProCon is more stable on contention rate, where standard deviations of them are 518.1 (ProCon) and 3635.8 (DS). As for 40-job experiment, the values are 1175 and 3713 for ProCon and DS respectively. ProCon is stable on balancing the resource contentions on workers.

Fig. 12 and Fig. 13 shows the CPU usage on each worker for ProCon and DS. Comparing them, Fig. 12 clearly tells that workloads are more evenly distributed in ProCon as the resource usage of CPU is in a similar level among all 4 workers. With DS, however, Worker-3 and 4 have lower load than Worker-1 and 2.

2) Lager cluster: Lastly, we increase the number of workers in the cluster. In the experiments, we use **Cluster 2**, which contains 1 Manger and 7 Workers. We conducted two experiments, one with 30 jobs that submitted to the cluster from 0 to 900s, the other one has 60 jobs with a submission interval of [0, 1200s].

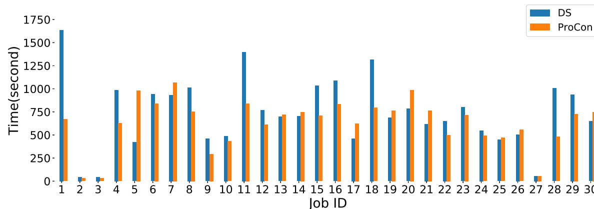


Fig. 14: Random schedule [0, 900s] with 30 random jobs

Completion Time and Makespan: The results for completion time are illustrated on Fig. 14 and Fig. 15. For the two experiments, there are 19 out of 30, and 36 out of 60 jobs that perceive improvement in ProCon with a degree of 31.5% and 35.9% reduction. Overall, completion time reduces 14.7% and 12.9% in the experiments respectively.

Besides, ProCon boosts the system by reducing the makespan. With the two experiments, values of makespan decrease 21.7% (1273.3s v.s. 1626.1s) and 37.4% (2811.0s v.s. 4493.5s) for ProCon and DS, respectively.

Contention Rate: ProCon achieves the improvement through efficiently routing the containers to the most desirable host, which balances the resource contention. Under a larger cluster with more workers, ProCon maintains a stable performance on the contention rate. For 30-job experiment, the standard deviation of contention rate is 588 for ProCon and 2045 for DS respectively. When the number of jobs increases, it is more difficult for the scheduler to allocate the jobs. With 40-job experiment, the values of standard deviation are 1792 and 6456 for ProCon and DS respectively. Clearly, ProCon enables a more stable container distribution for the cluster.

VII. Conclusion

This paper studies a progress based container placement scheme. In this paper, we propose ProCon, which not only considers instant resource utilization on the workers but also takes into account the estimation of future resource usage. We implemented ProCon on Kubernetes and conducted extensive experiments with deep learning applications. Comparing with default scheduler, ProCon achieves a significantly, up to 53.3%, reduction of completion time for a particular job and up to 23.0% across all. Additionally ProCon records an improvement of makespan for up to 37.4%.

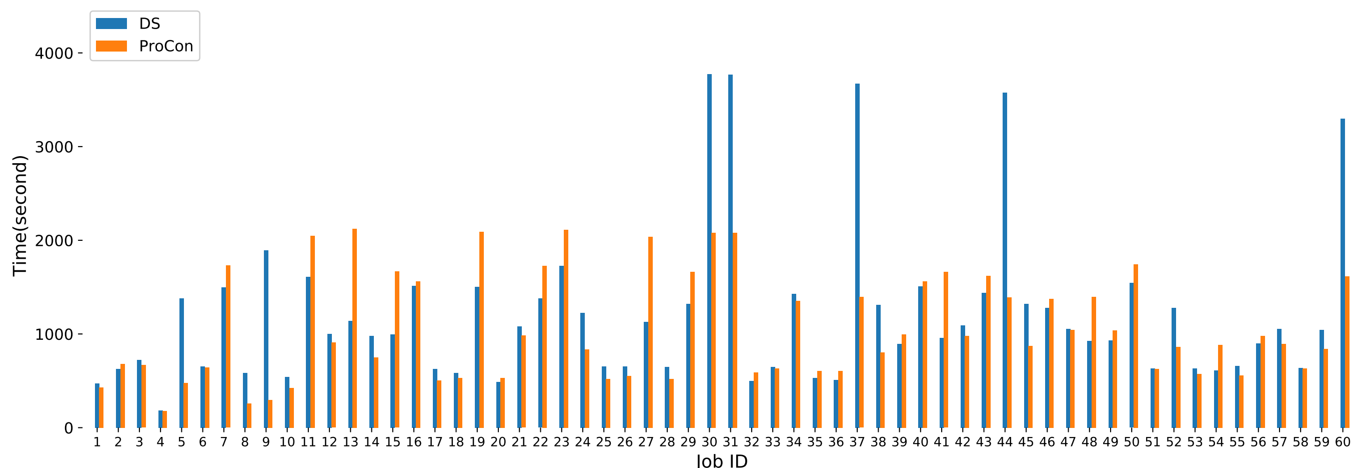


Fig. 15: Random schedule [0, 1200s] with 60 random jobs

REFERENCES

[1] Amazon personalize. https://aws.amazon.com/personalize/.

[2] Ibm: 5vs of big data. https://www.ibm.com/blogs/watson-health/the-5-vs-of-big-data/.

[3] Amazon web services. https://aws.amazon.com/.

[4] Sjaak Wolfert, Lan Ge, Cor Verdouw, and Marc-Jeroen Bogaardt. Big data in smart farming–a review. *Agricultural Systems*, 153:69–80, 2017.

[5] Mothive. https://www.mothive.com/.

[6] Cropx. https://www.cropx.com/.

[7] Arable. http://www.arable.com/.

[8] Economist report. https://www.economist.com/briefing/2017/05/06/data-is-giving-rise-to-a-new-economy.

[9] Microsoft azure. https://azure.microsoft.com/en-us/.

[10] Goole cloud platform. https://cloud.google.com/.

[11] Docker. https://docker.com/.

[12] Kubernetes. https://kubernetes.io/.

[13] Docker spread placement. https://docs.docker.com/engine/reference/commandline/service_create/.

[14] Balanced resource allocation. https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/algorithm/priorities/balanced_resource_allocation.go.

[15] Apache hadoop. https://hadoop.apache.org/docs/r3.0.0/.

[16] Tensorflow. https://www.tensorflow.org/.

[17] Cloud storage with aws. https://aws.amazon.com/products/storage/.

[18] Amazon elastic compute cloud. https://aws.amazon.com/ec2/.

[19] Azure data lake analytics. https://azure.microsoft.com/en-us/services/data-lake-analytics/.

[20] Azure machine learning service. https://azure.microsoft.com/en-us/services/machine-learning-service/.

[21] Containerization. https://cloud.google.com/containers/.

[22] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, page 1. ACM, 2016.

[23] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 181–195, 2016.

[24] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 251–263, 2017.

[25] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, number CONF, 2018.

[26] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *NSDI*, pages 79–94, 2019.

[27] Prajakta Kalmegh and Shivnath Babu. Mifo: A query-semantic aware resource allocation policy. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1678–1695. ACM, 2019.

[28] Wei Zhou, K Preston White, and Hongfeng Yu. Improving short job latency performance in hybrid job schedulers with dice. In *Proceedings of the 48th International Conference on Parallel Processing*, page 56. ACM, 2019.

[29] Diego Didona, Panagiota Fatourou, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. Distributed transactional systems cannot be fast. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 369–380. ACM, 2019.

[30] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. Comicon: A co-operative management system for docker container images. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 116–126. IEEE, 2017.

[31] Aditya Hegde, Rahul Ghosh, Tridib Mukherjee, and Varun Sharma. Scope: A decision system for large scale container provisioning management. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 220–227. IEEE, 2016.

[32] Docker swarm mode. https://docs.docker.com/engine/swarm/.

[33] Ying Mao, Jenna Oak, Anthony Pompili, Daniel Beer, Tao Han, and Peizhao Hu. Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2017.

[34] Andrew Chung, Jun Woo Park, and Gregory R Ganger. Stratus: cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 121–134. ACM, 2018.

[35] Fan Jiang, Kyle Ferriter, and Claris Castillo. Pivot: Cost-aware scheduling of data-intensive applications in a cloud-agnostic system.

[36] Wenjia Zheng, Michael Tynes, Henry Gorelick, Ying Mao, Long Cheng, and Yantian Hou. Flowcon: Elastic flow configuration for containerized deep learning applications. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 87:1–87:10, New York, NY, USA, 2019. ACM.

[37] Nsf cloudlab. https://www.cloudlab.us/.

[38] Pytorch. https://pytorch.org/.

[39] Vae. https://jaan.io/what-is-variational-autoencoder-vae-tutorial/.

[40] Mnist. http://yann.lecun.com/exdb/mnist/.

[41] Shuai Zheng and etc. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1529–1537, 2015.

[42] Mathias Berglund and etc. Bidirectional recurrent neural networks as generative models. In *Advances in Neural Information Processing Systems*, pages 856–864, 2015.

[43] Lenet-cnn. http://slazebni.cs.illinois.edu/spring17/lec01˙cnn˙architectures.pdf.