Advance Access publication on 31 May 2021
https://doi.org/10.1093/comjnl/bxab072

LraSched: Admitting More Long-Running Applications via Auto-Estimating Container Size and Affinity

Binlei Cai*[1], Qin Guo[2] and Junfeng Yu[3]

[1]*Shandong Computer Science Center (National Supercomputing Center in Jinan), Shandong Provincial Key Laboratory of Computer Networks, Qilu University of Technology (Shandong Academy of Sciences), China*
[2]*School of Science, Shandong Jianzhu University, China*
[3]*Igor Sikorsky Kyiv Polytechnic Institute, National Technical University of Ukraine, Ukraine*
**Corresponding author: adsert@126.com*

Many long-running applications (LRAs) are increasingly using containerization in shared production clusters. To achieve high resource efficiency and LRA performance, one of the key decisions made by existing cluster schedulers is the placement of LRA containers within a cluster. However, they fail to account for estimating the size and affinity of LRA containers before executing placement. We present LraSched, a cluster scheduler that places LRA containers onto machines based on their sizes and affinities while providing consistently high performance. LraSched introduces an automated method that leverages historical data and collects new information to estimate container size and affinity for an LRA. Specifically, it uses an online machine learning method to map a new incoming LRA to previous workloads from which we can transfer experience and recommends the amount of resources (size) and the degree of collocation (affinity) for the containers of the new incoming LRA. By means of recommendations, LraSched adapts the heuristic for vector bin packing to LRA scheduling and places LRA containers in a manner that both maximizes the number of LRAs deployed and minimizes the resource fragmentation, but without affecting LRA performance. Testbed and simulation experiments show that LraSched can improve the resource utilization by up to 6.2% while meeting performance constraints for LRAs.

Keywords: cloud computing; cluster scheduling; resource management; resource efficiency; quality of service.

Received 27 July 2020; Revised 10 March 2021; Editorial Decision 17 April 2021
Handling editor: Jin-Hee Cho

1. INTRODUCTION

An increasing number of long-running applications (LRAs) are now hosted in containerized data centers, such as user-facing services and database workloads. Unlike batch jobs that take from a few seconds to a few days to complete, LRAs should “never” go down, and handle short-lived latency-sensitive requests (a few μs to a few hundred ms) [41]. In containerized data centers, modern cluster managers leverage schedulers that allocate multiple resources (e.g., CPU and memory) to LRAs in the form of containers. As schedulers are application agnostic, modern cluster managers can consolidate different LRAs on a shared cluster, thereby achieving high cost efficiency.

Nevertheless, the majority of cluster facilities actually run at low utilization, which significantly reduces the cost efficiency [12, 30]. For example, Delimitrou et al. showed that a Twitter cluster hosting LRAs consistently achieved low CPU utilization (<20%) and low memory utilization (<50%) [13]. There are two primary reasons for the low utilization: (i) the LRA developers do not necessarily understand the resource requirements and submit oversized reservations to guarantee their performance goals (e.g., high throughput or low latency); and (ii) most cluster schedulers are oblivious to the degree of collocation of the containers of an LRA on the cluster nodes. They simply separate the containers of an LRA across different cluster nodes to avoid performance degradation due to resource contention, hence cannot tightly pack containers to the cluster.

Although there has been significant progress in cluster schedulers [4, 9, 22, 31, 36, 41], we are still facing challenges in improving resource utilization while guaranteeing LRA performance. First, it is difficult to determine the just-right amount of resources (size) for each LRA container to meet performance goals. Since the access load applied to an LRA varies with time, manually learning to build resource demand model for each LRA is not scalable. Second, precise control of LRA container placement is beneficial but difficult. A tight placement can create performance degradation due to the intra-node resource contention. A loose placement would create less contention, but exacerbate network latency and degrade cluster utilization [10, 16]. However, manually mining the best degree of collocation (affinity) of the containers for each LRA is difficult, since different LRAs have diverse behaviors. Third, when scheduling LRA containers with considering both multi-dimensional (i.e., multiple type) resources and placement constraints, the cluster scheduler is difficult to achieve global objectives.

In this work, we schedule LRAs based on adaptive resource provisioning and affinity-aware placement constraints. Our goal is to maximize the number of LRAs deployed and minimize the resource fragmentation, while meeting performance constraints for each LRA. To the best of our knowledge, there is no existing methods for such LRA scheduling problem. On the one hand, modern cluster managers like Borg [41], Mesos [22] and Kubernetes [4] cannot predict the amount of resources needed to meet LRA's performance constraints. They expect the LRA developers to provide resource reservation demands, and then schedule LRAs based on fairness. On the other hand, Quasar [13], Morpheus [24] and Elasecutor [28] can determine the amount of resources needed for each LRA based on historical data, but do not support for affinity-aware placement control. Medea [16] fully supports complex high-level placement constraints and global objectives. But it cannot automatically infer the most appropriate constraints for each LRA.

We present LRASched, a cluster scheduler that proactively provides resource reservations and affinity-aware placement constraints to LRAs and places LRA containers onto cluster nodes in an online manner. LRASched includes three key features. First, LRASched adaptively determines the container size for each LRA according to their performance constraints. Second, instead of empirically controlling the placement of containers, LRASched can infer the affinity-aware placement constraints automatically. Third, unlike existing approaches, LRASched leverages heuristics to pack the LRA containers onto cluster nodes based on both multi-dimensional resources and placement constraints, which aims to maximize the number of LRAs deployed and minimize the resource fragmentation. Our main contributions are summarized as follows:

(i) We present a dedicated scheduler for the placement of LRA containers, which can approximately achieve global objectives.

(ii) We create models to determine the container sizes and infer the affinity-aware placement constraints. First, we establish the profiles on the performance variances for previously scheduled LRA workloads. Second, by combining these profiling information with a minimal amount of profiling signals from the new incoming LRAs, we use online factorization machine (FM) to automatically recommend the container sizes and affinity-aware placement constraints needed to meet performance goals.

(iii) We design a two-phase heuristic scheduling solution to the APX-hard problem of placing LRA containers onto cluster nodes under multi-dimensional resources and affinity-aware placement constraints.

(iv) We implement LRASched on top of the Docker engine and evaluate its performance through simulations and a deployment on a 10-node cluster. Experimental results show that LRASched is able to increase the resource utilization by up to 6.2% while meeting performance constraints for LRAs, compared to previous scheduling approaches.

The rest of this paper is organized as follows. Section 2 first motivates our study of scheduling LRAs in shared clusters. The architecture and design of LRASched are described in Section 3, followed by the detailed evaluation of LRASched in both testbed and simulations is described in Section 4. Finally, we summarize the related work in Section 5 and conclude in Section 6.

2. MOTIVATION

In order to motivate the problem of scheduling LRAs in shared clusters, we first analyze the production trace from a production cluster in Alibaba. We then conduct a small-scale testbed experiment to further investigate the impact of the affinity-aware placement constraint on LRA performance.

2.1. Production workload analysis

We analyze a recently released trace dataset by Alibaba, which consists of both resource utilization and runtime information for LRAs and batch jobs in a 12-hour period [1]. Here, we focus on the data for LRAs to motivate the importance of LRA scheduling for the resource utilization. Alibaba cluster manager runs each LRA in a set of containers and reserves a fixed amount of resources for each container. We analyze the maximum ratio of used resources to reserved resources of each container over 12 hours as shown in Fig. 1a, b and c. We observe that most containers largely underutilize resources with regard to the amount they request. Specifically, Fig. 1d plots the

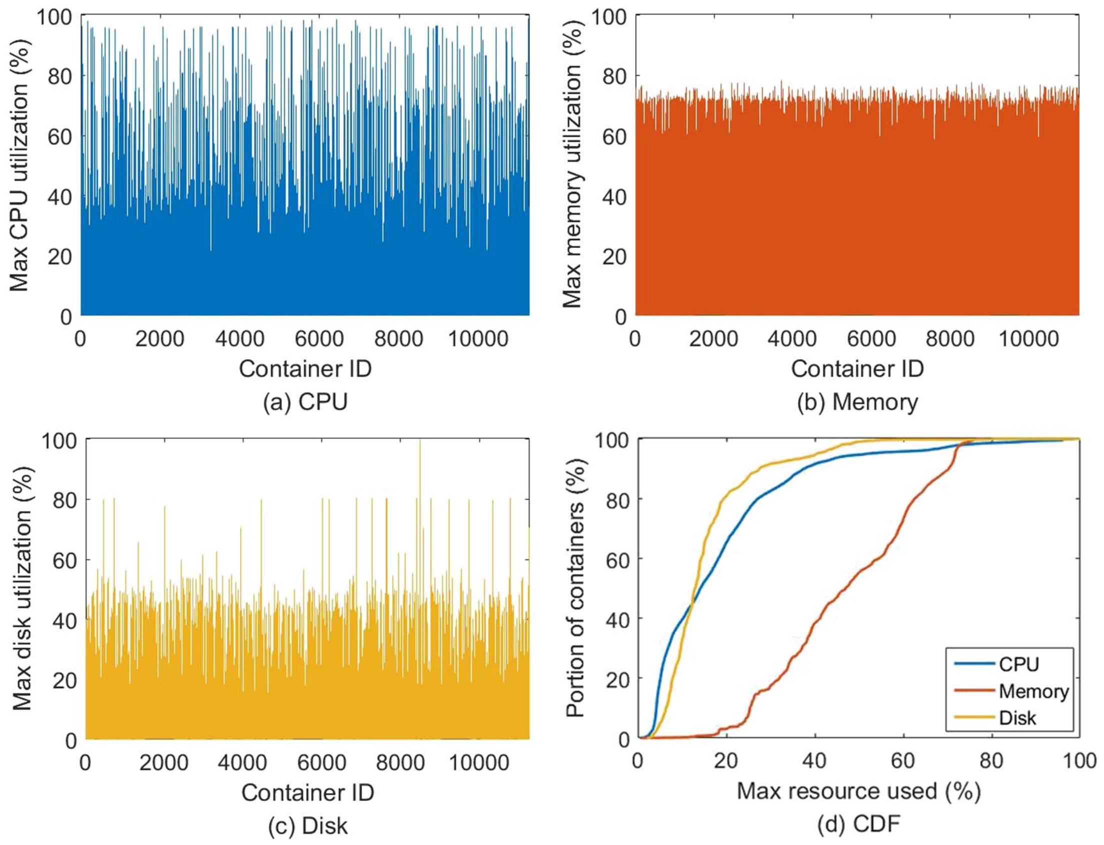


FIGURE 1. The analysis of resource utilization of containers in an Alibaba production cluster.

Cumulative Distribution Function (CDF) of containers' maximum CPU, memory and disk utilizations over 12 hours. There are 90% of containers whose maximum CPU and disk utilizations are lower than 40% and 28%, respectively. In contrast, memory utilization is a bit higher. Also, 90% of containers have a maximum memory utilization lower than 70%. *Observation: A large fraction of LRAs are over provisioned with excess resources. To improve cluster utilization, it is desirable to determine the right container size for each LRA under their given performance constraints.*

2.2. Impact of affinity-aware placement

We deploy a Cassandra instance composed of eight containers in a small-scale testbed (described in Sec. 4.1) and use YCSB benchmark [11] to generate load for this instance. We experiment with flexible affinity-aware placement constraints, which set different degrees of collocation of the containers on the cluster nodes. Figure 2 shows the performance variations of the Cassandra instance when we gradually vary the placement from anti-affinity constraint (i.e., one container per node) to full affinity constraint (i.e., all eight containers on one node). We observe that collocating two containers per node has a similar performance as the anti-affinity placement. It only degrades the throughput (or increases the 99th percentile latency) by about 1%. However, as the number of collocated containers increases, the performance of the Cassandra instance degrades significantly. Similarly, Panagiotis et al. presented performance analysis for a HBase instance with 10 containers, and showed that collocating 2 containers per node obtained the highest performance [16]. They also conducted experiments on TensorFlow workloads using 32 containers. The results showed that collocating four containers per node had similar runtime as the anti-affinity placement. *Observation: Simple anti-affinity placement constraint is beneficial, but is not sufficient. The tighter affinity-aware placement constraint is required to improve cluster utilization while guaranteeing LRA performance.*

3. ARCHITECTURE AND DESIGN

This section describes the architecture and key designs of LraSched.

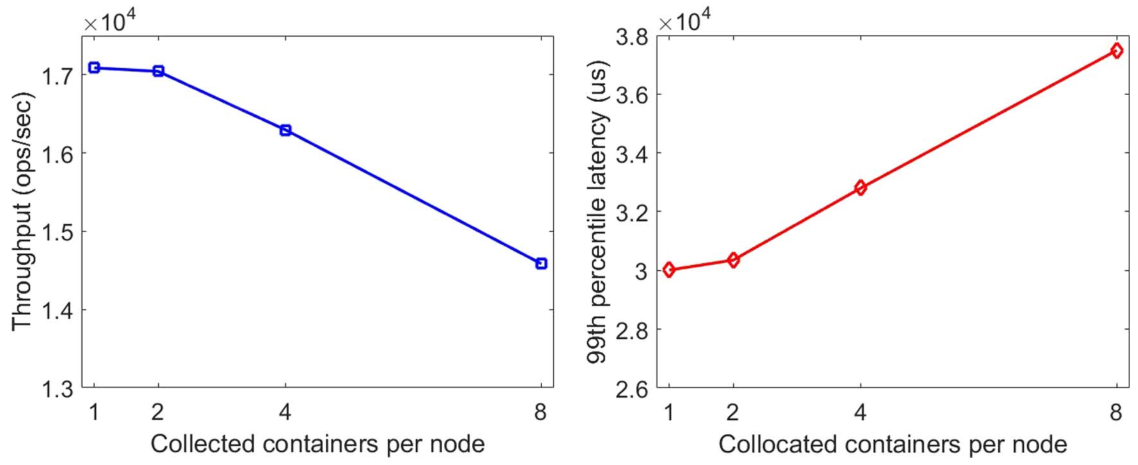


FIGURE 2. The impact of affinity-aware placement constraints on LRA performance.

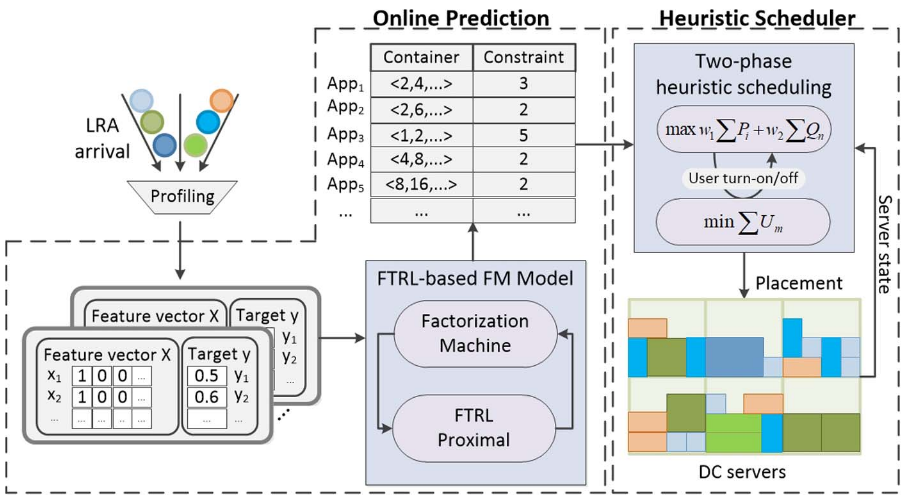


FIGURE 3. The architecture of LRASched.

3.1. Overview

Figure 3 illustrates the architecture of LRASched and its major components. LRASched is designed to operate in a container-enabled cluster where each LRA runs in a set of containers. It controls the placement of containers for LRAs and is basically composed of two components:

Online Prediction component quickly and accurately identifies the container size and affinity-aware placement constraint for each incoming LRA. It first profiles previously scheduled LRAs and transforms the large amount of profiling information into feature vectors. It then combines a minimal profiling information of incoming LRAs with these feature vectors, and uses the Follow-The-Regularized-Leader (FTRL)-based FM model to recommend the container sizes and affinity-aware placement constraints in a real-time manner.

Heuristic Scheduler component uses a two-phase heuristic scheduling model that determines the effective placement of containers of incoming LRAs based on the current cluster state. The scheduler adapts the heuristic for the vector bin packing problem to LRA scheduling wherein LRA container request arrivals in a streaming fashion. It accepts multiple LRA container requests at each scheduling period and packs them onto cluster nodes based on multi-dimensional resources and placement constraints, thereby attaining approximately global optimization objectives.

3.2. Online prediction

The key requirement for LRA scheduling is to quickly and accurately identify the best container size and affinity-aware placement constraint for an incoming LRA. Our goal is to perform online scheduling without any a priori knowledge about incoming LRAs. Most previous approaches address this issue with detailed but offline application characterization or long-term monitoring and modeling. Instead, we design a FTRL-based FM model to automatically identify the best container size and affinity-aware placement constraint for an incoming LRA in an online manner.

3.2.1. *FTRL-based FM model*

We first introduce the background of FM model, and then make the FM model work in online learning settings through incorporating the FTRL-Proximal algorithm [33].

FM model background. FM is a well-known supervised learning model working with any real-valued featured vectors [37]. It can define the prediction as a regression task over a set $X = \{\mathbf{x}_1, \mathbf{x}_2, ...\}$ ($\mathbf{x}_i \in \mathbb{R}^p$ is a feature vector with p real-valued variables), where a target function $\widehat{y} : X \to \mathbb{R}$ has to be estimated. For an input vector $\mathbf{x}_i$, FM predicts the result using the following formula,

$$\widehat{y}(\mathbf{x}_i) = w_0 + \sum_{l=1}^{p} w_l x_{i,l} + \sum_{l=1}^{p} \sum_{l'=l+1}^{p} x_{i,l} x_{i,l'} \sum_{f=1}^{h} v_{l,f} v_{l',f}, \quad (1)$$

where h is the number of dimensions of the factorization and $w_0 \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^p, \mathbf{V} \in \mathbb{R}^{p\times h}$ are the model parameters. Because Eq. (1) is equivalent to Eq. (2), the computational complexity of the FM model is $O(hp)$.

$$\widehat{y}(\mathbf{x}_i) = w_0 + \sum_{l=1}^{p} w_l x_{i,l} + \frac{1}{2}\sum_{f=1}^{h}[(\sum_{l=1}^{p} v_{l,f} x_{i,l})^2 - \sum_{l=1}^{p} v_{l,f}^2 x_{i,l}^2]. \quad (2)$$

The optimality of the FM model parameters is generally estimated by a loss function $l(\cdot)$. The optimal parameters are learned through minimizing the sum of the values of the $l(\cdot)$ over an observed set S.

$$\min \sum_{(\mathbf{x}_i, y_i)\in S} l(\widehat{y}(\mathbf{x}_i), y_i). \quad (3)$$

For regression, the most widely used loss function is square loss. Here, we formulate the regression least-squares loss as follows,

$$l(\widehat{y}(\mathbf{x}_i), y_i) = (\widehat{y}(\mathbf{x}_i) - y_i)^2. \quad (4)$$

Based on the loss function, several learning methods, such as stochastic gradient descent and alternating least-squares, have been proposed to learn the model parameters.

FM model refinement through online learning. Although FM model can be used to predict the container sizes and affinity-aware placement constraints for LRAs, majority of existing studies are conducted in the batch learning settings where all the training data is available before training [27]. However, in our case, the LRA requests arrive in a streaming fashion and the cluster scheduler should provide responses immediately. If the batch learning algorithms are applied in accordance with the streams in such scenario, the FM model has to be re-trained each time new LRA request arrives. This will incur high re-training cost. Therefore, we need to develop an efficient online learning algorithm for the FM model.

Many efforts have been made to produce online learning frameworks, such as FOBOS [14], RDA [42] and FTRL [33]. Among these frameworks, FTRL has been demonstrated to be more effective in production environments [26]. Therefore, we incorporate the FTRL with per-coordinate learning rates into FM. We use the FTRL to learn the FM model parameters in an online manner, and use the formula (1) to predict both the container sizes and affinity-aware placement constraints for incoming LRAs (**Algorithm 1**).

Algorithm 1 FTRL-based FM Model.

```
    Input: $\alpha^w, \alpha^v, \beta^w, \beta^v, \lambda_1^w, \lambda_1^v, \lambda_2^w, \lambda_2^v, \sigma, w_0 = 0, n_0^w = 0, z_0^w = 0 \forall l, w_l = 0, n_l^w = 0, z_l^w = 0, \forall l, \forall f, v_{l,f} \sim N(0,\sigma), n_{l,f}^v = 0, z_{l,f}^v = 0$.
    Output: $\widehat{y}(\mathbf{x}_i)$.
 1: for $i \in [1, T]$ do
 2:     Receive feature vector $\mathbf{x}_i$ and let $L = \{l | x_{i,l} \neq 0\}$;
 3:     Calculate $w_0$ using Eq.(5);
 4:     for $l \in L$ do
 5:         Calculate $w_l$ using Eq.(6);
 6:         for $f = 1$ to $h$ do
 7:             Calculate $v_{l,f}$ using Eq.(7);
 8:         end for
 9:     end for
10:     Calculate $\widehat{y}(\mathbf{x}_i)$ using the parameters calculated above;
11:     Update parameters using Algorithm 2;
12: end for
```

The FM model parameters are calculated by,

$$w_0 = \begin{cases} 0, & \text{if } |z_0^w| \le \lambda_1^w \\ -\dfrac{(z_0^w - \text{sgn}(z_0^w)\lambda_1^w)}{(\frac{\beta^w + \sqrt{n_0^w}}{\alpha^w} + \lambda_2^w)}, & \text{otherwise} \end{cases} \quad (5)$$

$$w_l = \begin{cases} 0, & \text{if } |z_l^w| \le \lambda_1^w \\ -\dfrac{(z_l^w - \text{sgn}(z_l^w)\lambda_1^w)}{(\frac{\beta^w + \sqrt{n_l^w}}{\alpha^w} + \lambda_2^w)}, & \text{otherwise} \end{cases} \quad (6)$$

$$v_{l,f} = \begin{cases} 0, & \text{if } |z_l^w| \le \lambda_1^w \\ -\dfrac{(z_{l,f}^v - \mathrm{sgn}(z_{l,f}^v)\lambda_1^v)}{(\frac{\beta^v + \sqrt{n_{l,f}^v}}{\alpha^v} + \lambda_2^v)}, & \text{otherwise,} \end{cases} \quad (7)$$

where α^w and β^w are the parameters for learning rate in the update of w_0 and w_l; α^v and β^v are the parameters for learning rate in the update of $v_{l,f}$; λ_1^w and λ_2^w are the $L1$ and $L2$ regularization for w_0 and w_l; λ_1^v and λ_2^v are the $L1$ and $L2$ regularization for $v_{l,f}$. The partial derivatives for the loss function are as follows,

$$g_0^w = \frac{\partial}{\partial w_0}(l(\widehat{y}(\mathbf{x}_i, y_i))) = 2(\widehat{y}(\mathbf{x}_i) - y_i)\frac{\partial}{\partial w_0}(\widehat{y}(\mathbf{x}_i)) \quad (8)$$

$$g_l^w = \frac{\partial}{\partial w_l}(l(\widehat{y}(\mathbf{x}_i, y_i))) = 2(\widehat{y}(\mathbf{x}_i) - y_i)\frac{\partial}{\partial w_l}(\widehat{y}(\mathbf{x}_i)) \quad (9)$$

$$g_{l,f}^v = \frac{\partial}{\partial v_{l,f}}(l(\widehat{y}(\mathbf{x}_i, y_i))) = 2(\widehat{y}(\mathbf{x}_i) - y_i)\frac{\partial}{\partial v_{l,f}}(\widehat{y}(\mathbf{x}_i)) \quad (10)$$

where

$$\frac{\partial}{\partial \theta}(\widehat{y}(\mathbf{x}_i)) = \begin{cases} 1, & \text{if } \theta = w_0 \\ x_{i,l}, & \text{if } \theta = w_l \\ x_{i,l}(\sum_{l=1}^{p} v_{l,f}x_{i,l} - v_{l,f}x_{i,l}^2), & \text{if } \theta = v_{l,f} \end{cases}$$

Algorithm 2 Parameter Update.

```
    Input: $\mathbf{x}_i, n_0^w, z_0^w n_l^w, z_l^w, n_{l,f}^v, z_{l,f}^v$.
 1: Calculate $g_0^w$ using Eq.(8);
 2: $\sigma_0^w = \frac{1}{\alpha^w}(\sqrt{n_0^w + (g_0^w)^2} - \sqrt{n_0^w})$;
 3: $z_0^w \leftarrow z_0^w + g_0^w - \sigma_0^w w_0$;
 4: $n_0^w \leftarrow n_0^w + (g_0^w)^2$;
 5: for $l \in L$ do
 6:     Calculate $g_l^w$ using Eq.(9);
 7:     $\sigma_l^w = \frac{1}{\alpha^w}(\sqrt{n_l^w + (g_l^w)^2} - \sqrt{n_l^w})$;
 8:     $z_l^w \leftarrow z_l^w + g_l^w - \sigma_l^w w_l$;
 9:     $n_l^w \leftarrow n_l^w + (g_l^w)^2$;
10:     for $f$ =1 to $h$ do
11:         Calculate $g_{l,f}^v$ using Eq.(10);
12:         $\sigma_{l,f}^v = \frac{1}{\alpha^v}(\sqrt{n_{l,f}^v + (g_{l,f}^v)^2} - \sqrt{n_{l,f}^v})$;
13:         $z_{l,f}^v \leftarrow z_{l,f}^v + g_{l,f}^v - \sigma_{l,f}^v v_{l,f}$;
14:         $n_{l,f}^v \leftarrow n_{l,f}^v + (g_{l,f}^v)^2$;
15:     end for
16: end for
```

3.2.2. *Identification for container size*

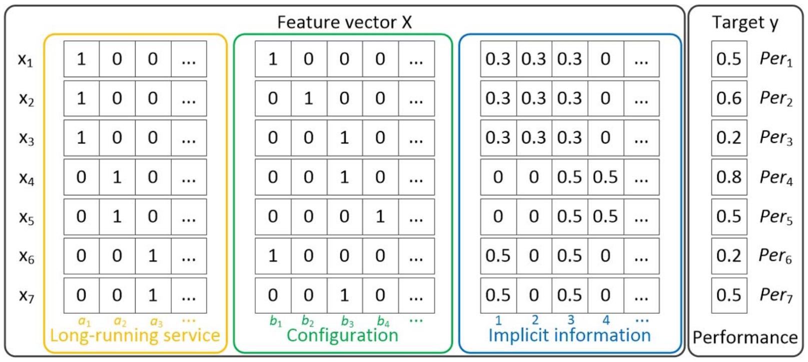
We use the FTRL-based FM model to identify how performance of an LRA varies with the different amount of resources reserved within a container.

Offline training. To improve the prediction accuracy, we first train the FTRL-based FM model using historical data. We select a small number of previously scheduled LRAs $A = \{a_1, a_2, ..., a_n\}$ (e.g., a few hundreds) and exhaustively profile them on different container configurations $B = \{b_1, b_2, ..., b_m\}$, where B is derived from the open-source data in both academia and industry. Here we only focus on CPU core and memory capacity in this work. We will incorporate network bandwidth in our future work. We perform profiling at a high-end platform that can provide the elastic container configurations. For an LRA a_ℓ in A, we use $Per_{a_\ell, b_\jmath}$ to denote the performance of a_ℓ when running on container configuration $b_\jmath$ (performance normalized between 0 to 1). The exhaustive profiling for LRA a_ℓ takes tens of minutes, but only needs to happen once. The profiling results can be represented by a set $D_1 = \{(a_\ell, b_\jmath, Per_{a_\ell, b_\jmath})\}$. We can easily transform the profiling results into feature vectors, and use them to train the FTRL-based FM model. For example, assume we have the profiling dataset D_1 as follows:

$$\begin{aligned} D_1 = \{&(a_1, b_1, 0.5), (a_1, b_2, 0.6), (a_1, b_3, 0.2), \\ &(a_2, b_3, 0.8), (a_2, b_4, 0.5), \\ &(a_3, b_1, 0.2), (a_3, b_3, 0.5)\} \end{aligned}$$

Figure 4 illustrates an example of how to generate feature vectors from the profiling dataset. Each row represents a feature vector $\mathbf{x}_i$ with its corresponding performance target y_i (e.g., tail latency). Here, the first $|A|$ binary indicator variables (yellow) are used to indicate the active LRA of a profiling—there exists only one active LRA in each profiling. For example, $(1, 0, 0, ...)$ represents the LRA a_1. The next $|B|$ binary indicator variables (green) are used to indicate the active container configuration—again there exists only one active container configuration in each profiling. For example, $(1, 0, 0, 0, ...)$ represents the container configuration b_1. The feature vectors also include $|B|$ indicator variables (blue) that are used to indicate all the container configurations the active LRA has been profiled on. For example, $(0.3, 0.3, 0.3, 0, ...)$ represents the LRA a_1 has been profiled on container configuration b_1, b_2, b_3. The target Per_1 represents the corresponding performance of the LRA a_1 running on the container configuration b_1. Since the number of LRAs in our scenario is not constant, we must incorporate new LRAs into the current model somehow. Using an idea similar to [25], we take a simple approach that a zero and random vector are respectively inserted into $\mathbf{w}$ and $\mathbf{V}$ as the initial parameters of the new features.

Online predicting for container size. For an incoming LRA a_i, we first profile it briefly with two randomly selected container configurations from B and transform profiling results


Feature vector X
Target y
Long-running service
Configuration
Implicit information
Performance

	a_1	a_2	a_3	...	b_1	b_2	b_3	b_4	...	1	2	3	4	...	Target y	
x_1	1	0	0	...	1	0	0	0	...	0.3	0.3	0.3	0	...	0.5	Per_1
x_2	1	0	0	...	0	1	0	0	...	0.3	0.3	0.3	0	...	0.6	Per_2
x_3	1	0	0	...	0	0	1	0	...	0.3	0.3	0.3	0	...	0.2	Per_3
x_4	0	1	0	...	0	0	1	0	...	0	0	0.5	0.5	...	0.8	Per_4
x_5	0	1	0	...	0	0	0	1	...	0	0	0.5	0.5	...	0.5	Per_5
x_6	0	0	1	...	1	0	0	0	...	0.5	0	0.5	0	...	0.2	Per_6
x_7	0	0	1	...	0	0	1	0	...	0.5	0	0.5	0	...	0.5	Per_7

FIGURE 4. Example for the set S that is created from the example dataset D_1.

as new feature vectors. The process of profiling only takes tens of seconds, which is acceptable for LRAs. Second, by combining the feature vectors of both previously scheduled LRAs and new incoming LRA, we use FTRL-based FM model to predict the missing targets across all other container configurations. We select the configuration that just leads to achieve the performance goal as the container size $b_i = \{b_{ij}^k\}$ for the LRA a_i, where b_{ij}^k denotes the resource demand of the container j of LRA a_i in dimension k.

3.2.3. *Identification for affinity-aware placement constraint*

We also use FTRL-based FM model to identify how performance of an LRA varies with the different degrees of collocation of the containers on the cluster nodes.

Offline training. We profile the LRAs $A = \{a_1, a_2, ..., a_n\}$ offline to derive their performance on each container size under different collocation degrees $C = \{c_1, c_2, ..., c_m\}$, where $c_1 = 2^0, c_2 = 2^1, ..., c_m = 2^{m-1}$[1]. For an LRA a_ℓ in A, we profile it with c_ι collocated containers per container size. We use $Per_{a_\ell,b_\jmath,c_\iota}$ to denote the performance of a_ℓ when running on container size $b_\jmath$ under collocation degree c_ι. The profiling results can be represented by a set $D_2 = \{(a_\ell, b_\jmath, c_\iota, Per_{a_\ell,b_\jmath,c_\iota})\}$. We also can transform the profiling results D_2 into feature vectors, and use them to train the FTRL-based FM model.

Online prediction for affinity-aware placement constraint. For an incoming LRA a_i, we first profile it for tens of seconds with two randomly-selected collocation degrees under container size b_i and transform profiling results as new feature vectors. Second, by combining the feature vectors of both previously scheduled LRAs on container size b_i and new incoming LRA, we use FTRL-based FM model to predict the missing targets across all other collocation degrees. We select the collocation degree that just leads to achieve the performance goal as the affinity-aware placement constraint lim_i for the LRA a_i, where $lim_i \in C$.

3.3. Heuristic scheduler

We develop a heuristic scheduler that can place LRA containers onto cluster nodes effectively. The scheduler is invoked periodically, and first needs to receive the following inputs: LRA requests and affinity-aware placement constraints from each newly submitted LRA, and idle resources at each cluster node. Then, during the latest interval, it adopts a two-phase scheduling policy to determine the cluster node where each LRA container should be placed. In Phase 1, it tries to (i) consider the multi-dimensional resources of both LRA containers and cluster nodes, (ii) place all the containers of an LRA without violating its affinity-aware placement constraint and (iii) achieve approximately global optimization objectives. If some LRA requests are not successfully scheduled, the user can choose to whether to start Phase 2, which aims to minimize the number of nodes added to place unscheduled LRA containers.

3.3.1. *Two-phase scheduling model*

Placing LRA containers onto cluster nodes is analogous to vector bin packing, where LRA containers correspond to multi-dimensional items and cluster nodes correspond to bins. The main difference is that the cluster nodes have variable sizes. Here we model the two-phase LRA scheduling based on vector bin packing through adding some additional placement constraints. Below, we give an intuitive description, relying on the notations from Table 1.

Phase 1. We assume that M LRAs are submitted to a cluster comprised of N nodes in current scheduling period.

$$\max \quad w_1 \sum_{i=1}^{M} P_i + w_2 \sum_{n=1}^{N} Q_n \tag{11}$$

$$\text{s.t.} \quad \forall\, i, j : \sum_{n=1}^{N} X_{ijn} \le 1 \tag{12}$$

TABLE 1. Definition of Notations.

Symbol	Description
M	The number of LRAs submitted in the latest scheduling period
N	The number of nodes in the current cluster
G_i	The number of container requests of LRA a_i
F_n^k	The amount of free resource left by node n in dimension k
I_n	A large enough integer used in Eq.(16)
P_i	A 0-1 variable, 1 if all the containers from LRA a_i can be successfully placed, 0 otherwise
X_{ijn}	A 0-1 variable, 1 if the container j of LRA a_i is placed on node n, 0 otherwise
lim_i	The degree of collocation of the containers for LRA a_i
b_{ij}^k	The resource demand of the container j of LRA a_i in dimension k
r_{min}^k	The minimal resource demand in dimension k
q_n^k	A 0-1 variable, 1 if the free resource of node n in dimension k is larger than r_{min}^k after container placement, 0 otherwise
Q_n	A 0-1 variable, 1 if free resource of node n in every dimension is larger than the minimum resource demand, 0 otherwise
V	Number of resource dimensions
U_m	A 0-1 variable, 1 if node m is added, 0 otherwise

$$\forall\, i, n : \sum_{j=1}^{G_i} X_{ijn} \leq lim_i \tag{13}$$

$$\forall\, n, k : \sum_{i=1}^{M}\sum_{j=1}^{G_i} b_{ij}^k \cdot X_{ijn} \leq F_n^k \tag{14}$$

$$\forall\, i : \sum_{n=1}^{N}\sum_{j=1}^{G_i} X_{ijn} - G_i \cdot P_i = 0 \tag{15}$$

$$\forall\, n, k : \sum_{i=1}^{M}\sum_{j=1}^{G_i} b_{ij}^k \cdot X_{ijn} - I_n(1 - q_n^k) \leq F_n^k - r_{min}^k \tag{16}$$

$$\forall\, n : \sum_{k=1}^{V} q_n^k - V \cdot Q_n = 0 \tag{17}$$

Objective: Our objective function (11) consists of two components: (i) the first part is to maximize the number of successfully scheduled LRAs and (ii) the second part aims at reducing resource fragmentation in the cluster. We can meet diverse desired cluster behaviors by assigning different weights (i.e., w_1 and w_2) to these two components.

Placement constraints: To achieve above objective, we need to satisfy six placement constraints: Eq. (12) implies that each container of an LRA should be placed to only one node. Eq. (13) implies that the degree of collocation of the containers of LRA a_i on each node is no more than lim_i. Eq. (14) implies that the resource demand of containers assigned to node n in dimension k is no more than the free resources of node n. Eq. (15) implies that we must place either all or none of an LRA's containers. Eq. (16) uses the binary indicator variable q_n^k to indicate whether the left resources of node n in dimension k is larger than the minimum resource demand. We choose the smallest container size from $B = \{b_1, b_2, ..., b_m\}$ as the minimum resource demand r_{min}^k. Eq. (17) uses the binary indicator variable Q_n to indicate whether the left resources of node n in every dimension is larger than the minimum resource demand.

Phase 2. The Phase 2 of the scheduler begins to scale resources up through adding nodes to place LRA containers that are not successfully placed in Phase 1. Its objective is to minimize the number of added nodes, as shown in function (18). Besides the constraint Eqs. (12) and (15), the objective is subjected to all the placement constraints similar as in Phase 1. The changed constraint is used to restrict that each container has to be placed onto one of the newly added nodes, as shown in Eq. (19). After Phase 2, all LRA containers can be successfully placed onto cluster nodes.

$$\min \sum_{m=1}^{N'} U_m \tag{18}$$

s.t. (13), (14), (16), (17)

$$\forall\, i, j : \sum_{m=1}^{N'} X_{ijm} = 1 \tag{19}$$

Based on the formulations of both **Phase 1** and **Phase 2**, we find that, like vector bin packing, the problem of placing multi-dimensional containers of LRAs onto cluster nodes is APX-hard even when placement constraints are eliminated [15, 18]. Although we can formulate the problem as an Integer Linear Programing (ILP) model and then solve the model via ILP solvers (e.g., CPLEX solver), it has been the experience of others (e.g., [35, 39]) that these solvers do not scale beyond smallish LRAs. Therefore, we present a heuristic solution to place the multi-dimensional containers of LRAs onto cluster nodes under placement constraints.

3.3.2. *Heuristic-based scheduling algorithm*

To design an efficient heuristic scheduling algorithm, we first systematically study the solution in a one-dimensional space. An efficient heuristic proceeds by repeatedly matching the biggest container that fits into the smallest feasible node under satisfying all placement constraints. Intuitively, this method is able to reduce the resource fragmentation in each node, thereby increasing the number of successfully scheduled LRAs.

Measure. In order to extend above method to multi-dimensional problems, we need to define an ordering on LRAs. Here, we define the ordering by a measure, which is a size function and can return a scalar for each LRA. For LRA a_i, we define its measure as follows,

$$W_i = \sum_{k=1}^{V} \frac{b_i^k}{b^k}, \tag{20}$$

where $b_i^k = \sum_{j=1}^{G_i} b_{ij}^k$ denotes the total resource demand of LRA a_i in dimension k, and $b^k = \sum_{i=1}^{M} \sum_{j=1}^{G_i} b_{ij}^k$ denotes the total resource demand of M LRAs in dimension k.

Fitness. In addition, we also need to define the *fitness* of a container relative to a node across multiple resource dimensions. The larger fitness implies the lower the fragmentation. Using the similar idea in [35], we use the dot product to measure the fitness of container j of LRA a_i relative to the node n as follows,

$$f_{ijn} = \sum_{k=1}^{V} \frac{b_{ij}^k}{b^k} \cdot \frac{F_n^k}{F^k}, \tag{21}$$

where F_n^k denotes the free resource of node n in dimension k, and $F^k = \sum_{n=1}^{N} F_n^k$ denotes the total free resource of all nodes in dimension k. The fitness has been demonstrated to provide the best packing efficiency [15, 18, 35].

Heuristic algorithm. In Phase 1, the scheduler first sorts the LRAs in decreasing order of their measures. Starting with the first LRA, the scheduler selects a container within it, and selects a set of nodes, each of which satisfies the container's placement constraints. If the set is not empty, the scheduler computes the container's fitness relative to each node of the set, and selects the node with highest fitness as the candidate placement node. If the set is empty, the scheduler interrupts the processing of this LRA. This process is repeated recursively until all the containers of this LRA has been processed or an interrupt occurs. If each container of this LRA has candidate placement node, the scheduler will schedule this LRA, otherwise it will not scheduled. Then, the scheduler proceeds to the second LRA, repeating the same process.

If the user chooses to start Phase 2, the scheduler adds a new node. It selects a set of unplaced LRA containers whose placement constraints can be satisfied on this node. For each LRA container in the set, it computes the fitness relative to this node. The LRA container with the highest fitness is placed. This process is repeated recursively until the node cannot accommodate any further LRA containers. The scheduler then considers this node to be filled and adds the next new node, repeating the same process until no LRA containers need to be placed. The complete algorithm is sketched in **Algorithm 3**.

4. EVALUATION

We implement a prototype of LraSched using Python. To validate the effectiveness of LraSched, we compare it with two other LRA scheduling schemes through testbed and simulation experiments.

4.1. Experimental setup

Testbed. A series of realistic experiments are conducted on a shared cluster comprised of 10 servers. Each server is equipped with two Intel Xeon E5-2620 8-core CPU, 32 GB RAM, running Ubuntu 16.04. All servers are connected via a top-of-rack switch. LraSched prototype acts as an extension to the scheduling component of Apache Yarn-2.7.3 [40]. Docker-18.09 [3] is used to create OS containers.

Simulation. To evaluate LraSched under multiple parameter choices (e.g., LRA scales, resource dimensions), we employ a simulator that executes LraSched using simulated servers. In the process of simulation, we only ignore the RPCs and task execution.

Workloads. We use Cassandra, HBase, Memcached and Apache webserver to mimic LRAs. By varying the input dataset and incoming load distribution, we can mimic various LRA workloads.

Alternative LRA Scheduling Schemes. To validate the benefit of LraSched, we compare it to two other LRA scheduling schemes:

- ILP-based scheme: It formulates the container placement of LRAs with affinity-aware placement constraints as an ILP model, and solves it using a standard ILP solver.
- FFD-based scheme: Similar to [15, 18, 35], it uses the First Fit Decreasing (FFD)-based heuristic algorithm to place the containers of LRAs, but with anti-affinity constraints.

4.2. Experimental results

In our experiments, we focus on answering three key questions: (i) the accuracy of the FTRL-based FM model within LraSched, (ii) the impact of LraSched on cluster utilization and LRA performance and (iii) the scalability of LraSched.

4.2.1. *Testbed results*

To evaluate LraSched in a real environment, we deploy 10 Cassandra, 10 HBase, 10 Memcached and 10 Apache webserver instances on the small-scale testbed described above. We turn off the Phase 2 in the two-phase heuristic algorithm of LraSched.

Accuracy validation for FTRL-based FM model. We use RMSE (Root Mean Square Error) to evaluate the errors between our prediction results and the real values. RMSE is the root mean squared error of the prediction [19]. Figure 5 shows the experimental results for the validation of the accuracy of FTRL-based FM model. From our observation, our

Algorithm 3 Two-phase Heuristic Scheduling.

```
    Input:$N$ nodes list $LN$, $M$ LRAs list $LA = \{a_i\}$.
    Output: The placement for each container.
 1: Phase 1:
 2: Sort $\{a_i\}$ in decreasing order of their resource demands according to their measures;
 3: for $i = 1$ to $M$ do
 4:     while The container list $LC_i$ of $a_i$ is not empty do
 5:         Select a container $c_j$;
 6:         Select a set of nodes from $LN$, each of which satisfies $c_j$'s placement constraints (Eq.(13), (14), (16), (17));
 7:         if The set is not empty then
 8:             Compute the fitness of $c_j$ relative to each node;
 9:             Select the node $n$ that has the highest fitness;
10:             Insert the pair $< c_j, n >$ into $a_i$'s scheduling plan;
11:             Remove $c_j$ from $LC_i$;
12:         else
13:             break;
14:         end if
15:     end while
 6:     if The list $LC_i$ is empty then
17:         Place $a_i$' containers according to its scheduling plan;
18:         Remove $a_i$ from $LA$;
19:     else
20:         $a_i$ is not scheduled and $LC_i$ is reset to its initial state;
21:     end if
22: end for
23: User turn-on/off
24: Phase 2:
25: Merge remaining LRA container requests into list $LC$;
26: while $LC$ is not empty do
27:     Add a new node $n^*$;
28:     while There are unplaced containers that fit into $n^*$ do
29:         Compute the fitness of each container relative to $n^*$;
30:         Place the container that satisfies placement constraints (Eq.(13), (14), (16), (17)) and has the highest fitness;
31:         Remove the container from $LC$;
32: :   end while
33: end while
```

FTRL-based model achieves high prediction accuracy. On average, RMSE is less than 0.25 across all workloads, while maximum RMSE is less than 0.5, ensuring that the predictions that drive LraSched decisions are accurate. Moreover, The RMSE value shows downward trend significantly with the running time. Moreover, we also evaluate the performance of our FTRL-based FM model and the matrix factorization (MF) model used in [13]. As shown in Table 2, FTRL-based FM model achieves higher prediction accuracy than the online MF model. Although prediction latency of FTRL-based FM model is higher than online MF model, it is sufficient to satisfy practical time requirements for LRA scheduling.

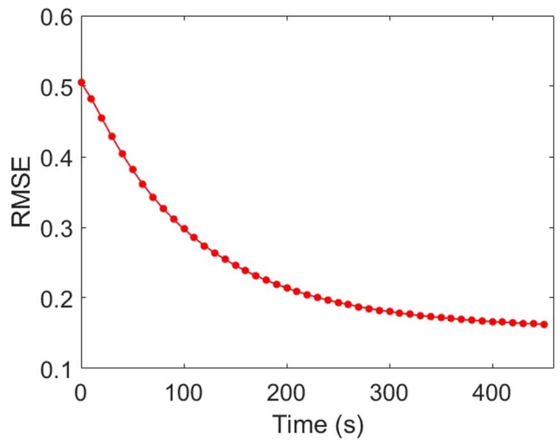


FIGURE 5. Validation of LraSched's FTRL-based FM model.

TABLE 2. The results of prediction accuracy and latency.

Model	RMSE	Latency (ms)
FTRL-based FM	0.243	1.15
Matrix factorization	0.582	0.73

Utilization and performance evaluation. We define the utilization as $\eta = 1 - \frac{R_{free}}{R_{total}}$, where R_{total} represents the total amount of resources in the cluster, and R_{free} represents the amount of unallocated resources in the cluster. Figure 6 shows the changes of the testbed utilization with running time. We observe that the utilization of the testbed has the same change trends under these three schemes before 200 seconds. This is mainly because all schemes can successfully schedule the new incoming LRAs. As the available resources of the testbed decrease, there exist some LRAs that cannot be scheduled successfully. However, our scheme can pack containers tightly, and can successfully schedule almost as many LRAs as ILP-based scheme. Therefore, it achieves higher resource utilization. In contrast, the FFD-based scheme results in a slightly larger number of LRAs that cannot be successfully scheduled, due to anti-affinity constraints. We further study the impact of LraSched on LRA performance. We use tail latency as the metric to evaluate the performance of LRAs. A tail latency SLO (Service Level Objective) such as a 99th percentile of

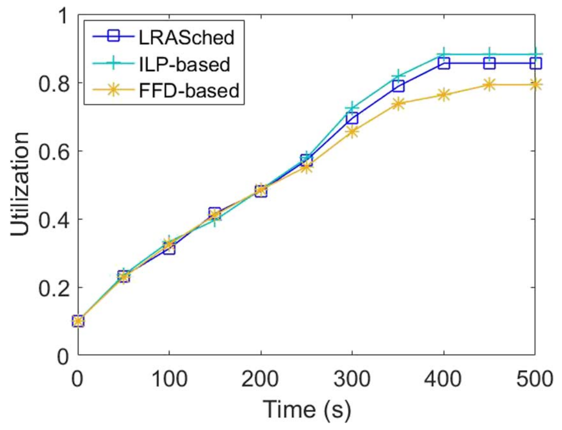


FIGURE 6. The changes of cluster utilization over running time.

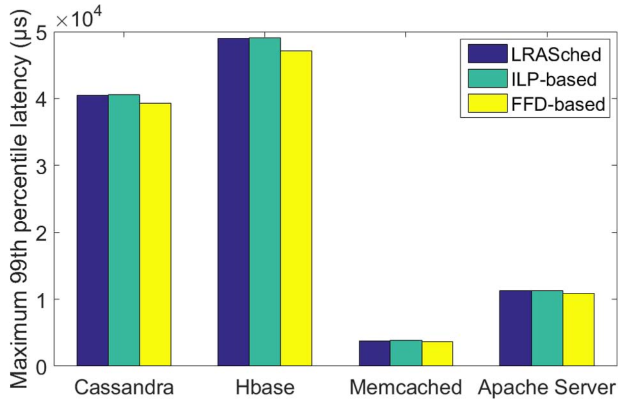


FIGURE 7. Application performance comparison.

50ms requires that 99% of requests complete within 50ms. Figure 7 shows the maximum 99th percentile latencies of Cassandra, HBase, Memcached and Apache webserver instances, respectively. It is easy to see that LraSched and ILP-based scheme have the similar impact on LRA performance. This is mainly because ILP-based scheme uses the affinity-aware placement constraints that are derived from the online prediction component of LraSched. FFD-based scheme slightly outperforms LraSched and ILP-based scheme across all workloads due to its anti-affinity placement constraint. However, the 99th percentile latencies of LraSched only increase on average by 3.2%, 3.9%, 2.7% and 3.5% for Cassandra, HBase, Memcached and Apache webserver, respectively

4.2.2. *Simulation results*

We also use a simulated cluster to conduct experiments for these three scheduling schemes. To mimic the realistic state of machines in large-scale shared clusters, we generate the simulated machines with different sizes. To ease description, we normalize the size of each resource dimension of the simulated machines to $[0.1, 1]$ and randomly draw the size in each resource dimension from the range $[0.1, 1]$ using a uniform distribution. Similarly, to mimic various types of LRAs, the container sizes of LRAs are chosen independently using a uniform distribution on $(0, 0.5]$.

Utilization evaluation. At the beginning, we turn off the Phase 2 in the two-phase heuristic algorithm. We first generate 500 LRAs, each of which runs in a set of containers. We randomly and independently set a limit (ranging in [1,10]) on the number of collocated containers for each LRA. We submit these LRAs to a cluster with 100 simulated machines. We vary the number of resource dimensions in the set of values $\{3, 4, 5, 6\}$. Figure 8a shows the percentage of LRAs that are scheduled successfully under different schemes when considering different resource dimensions. We find that the ILP-based scheme successfully schedules more than 86.1% of the LRAs in average. Since it solves the ILP using the ILP solver, this scheme has poor scalability with the cluster size scaling up (Figs 11 and 12). Our scheme gives a similar average result. However, it has a better online scalability, as shown in Figs 11 and 12. The FFD-based scheme gives an average of 75.5% of successfully-scheduled LRAs due to the anti-affinity constraints. We further analyze the percentage of successfully-scheduled LRAs under different scales of submitted LRAs. We generate 1000 and 2000 LRAs, and submit them to cluster with 300 and 500 simulated machines, respectively. We find that our scheme is leading to high ratio of successfully-scheduled LRA across all scales (Fig. 8b and c). Subsequently, we analyze the resource utilizations of the cluster when these schemes schedule LRAs across different resource dimensions and scales (Fig. 9). It is easy to find that the ILP-based scheme achieves better resource utilization (about 87.5%) than the other ones. Because our scheme can schedule almost as many LRAs as the ILP-based scheme, it also can achieve higher utilization (about 86%). The FFD-based scheme has the poor results (about 79.8%), since there are existing a fraction of LRAs that cannot be scheduled on the cluster successfully.

We then turn on the Phase 2. That is, we increase the number of the simulated machines until all the submitted LRAs can be scheduled successfully under different schemes. Figure 10a shows the number of simulated machines used by each of the schemes for 500 submitted LRAs. We observe that our scheme's behavior is very similar to ILP-based scheme, and both of them consistently outperform the FFD-base scheme in all dimensional cases. For example, the average number of simulated machines used by our scheme and ILP-based scheme are 116, 114 respectively. However, FFD-based scheme uses 125 simulated machines in average. However, our scheme has better performance than ILP-based scheme in case of large scale submitted LRAs (Figs 11 and 12). Figure 10b and c compare these three schemes on other two scales of submitted LRAs. It is easy to see that our scheme nearly always does as

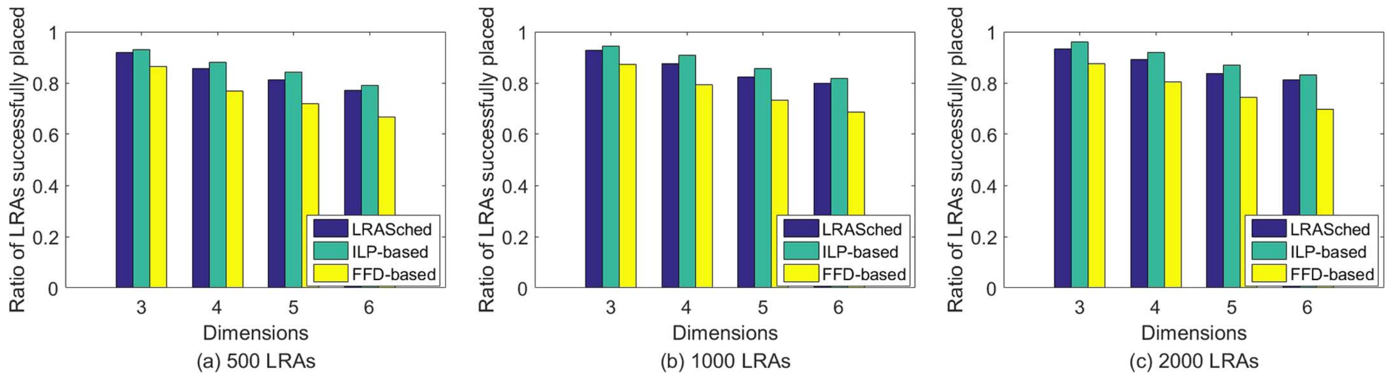


FIGURE 8. The percentage of LRAs successfully scheduled under various submitted LRA scales.

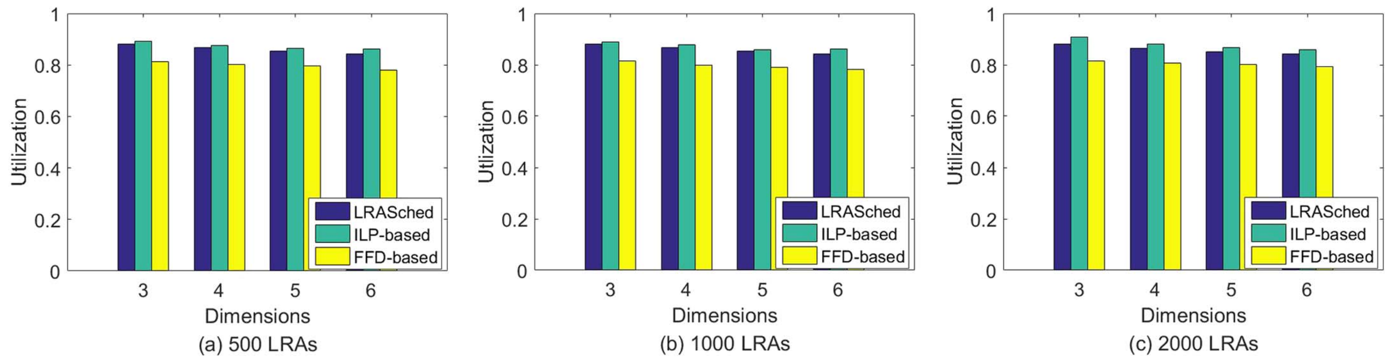


FIGURE 9. The utilization of cluster under various submitted LRA scales.

well as ILP-based scheme, and often noticeably outperforms FFD-based scheme.

Scalability evaluation. We study the scalability of all schemes by varying the size of the simulated cluster. Figures 11 and 12 show the average latency to place all the containers of an LRA under 3- and 6-dimensional cases. The FFD-based scheme achieves the lowest latencies. For large clusters, its highest latency is lower than 40 ms. Our heuristic scheme leads to slightly higher latencies, due to the affinity-aware placement constraints and dot product computing. In the worst case, the latency of our heuristic scheme only reaches up to 170 ms (Fig. 12). ILP-based scheme has the highest latency. As an optimization problem, the scheduling latency of ILP-based increases quickly as the size of the cluster scales up. For small cluster size, it has similar latency to both FFD-based and LraSched schemes. However, its latency reaches up to 890 ms for large cluster size.

5. RELATED WORK

We discuss work relevant to LraSched in the areas of cluster manager, virtual machine (VM) scheduling and interference-aware scheduling.

Cluster management frameworks. Most cluster managers, such as Borg [41], Mesos [22], Omega [38] and Yarn [40], support LRA management in shared clusters. Borg [41] uses a logically centralized component to schedule both LRAs and short running applications. Mesos [22] and Omega [38] employ two-level schedulers that request resources for each LRA from a central resource manager and make them execute over these resources in a coordinated fashion. Slider [2] and ongoing extension efforts [5] extend Yarn [40] to support LRAs. However, all these managers make resource reservations for LRAs according to the resource requests of LRA developers. They cannot adaptively identify the just right amount of resources needed to satisfy LRA's performance goal. Quasar [13], Morpheus [24] and Elasecutor [28] can determine the amount of resources needed for each LRA based on historical data. According to users' performance constraints for each LRA, Quasar [13] uses classification techniques to predict the amount of resources needed to satisfy these constraints at any point. Morpheus [24] and Elasecutor [28] allocate time-varying resources to LRA according to historical resource usage patterns. However, all of them do not consider the degree of collocation of the containers on the cluster nodes when scheduling LRAs. There exist a few schedulers that

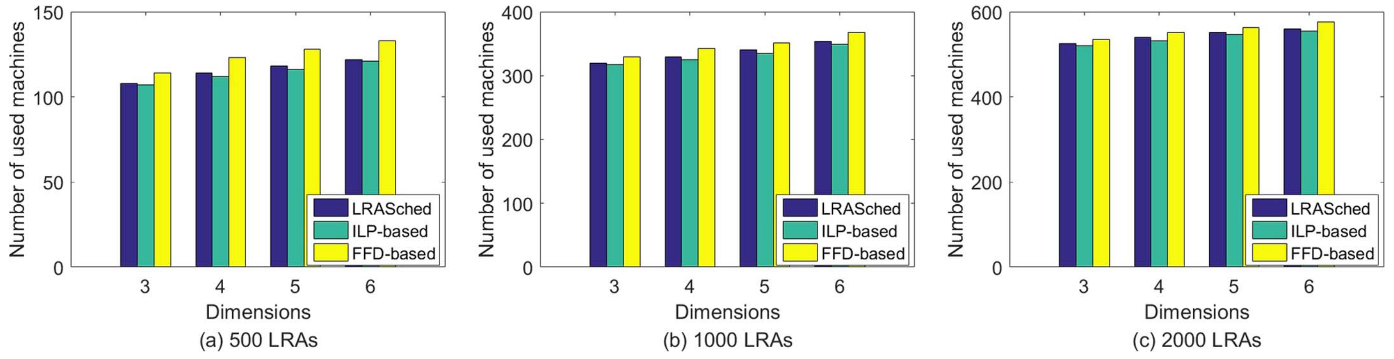


FIGURE 10. The number of used machines under various submitted LRA scales.

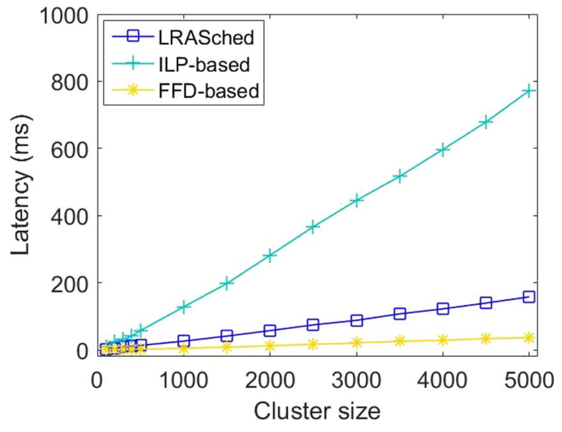


FIGURE 11. Scheduling latency of different schemes with three dimensions under various cluster scales.

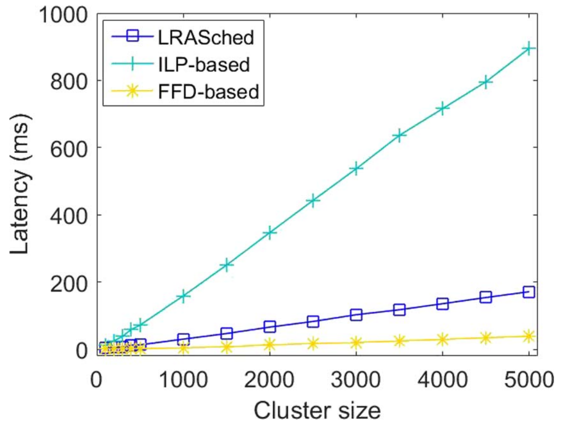


FIGURE 12. Scheduling latency of different schemes with six dimensions under various cluster scales.

preliminarily support container placement constraints [4, 16, 17]. Firmament [17] employs a graph-based approach to support simple placement constraints, such as placing applications onto specific machines. Kubernetes [4] and Meada [16] support explicit affinity-aware placement constraints to express dependencies between containers. But the constraints used in these schedulers are decided empirically. Both of them cannot automatically infer the most appropriate constraints for each LRA. To match the characteristics of BSP (Bulk Synchronous Parallel) jobs, SPIN [20] jointly considers the BSP job placements and scheduling so as to minimize the period of time to finish all BSP jobs. Zhang et al. propose an online scheduling algorithm that can decide the the number/size of container and running time for each machine learning job, with the goal of minimizing the average completion time [45]. Unlike BSP and machine learning jobs whose critical performance is completion time, LRAs focuses on request latency distribution and has strict performance goal (e.g., tail latency SLO). So the above two approaches cannot be applied to LRA scheduling directly.

VM scheduling. LRA container scheduling has some similarities with VM scheduling, since VMs are usually long running. Consolidating VMs that have multi-dimensional resource demands onto machines can be translated into the vector bin packing problem [34]. Therefore, the scheduling problem solved by LraSched is a significant generalization of online vector bin packing problem [6–8, 21]. By means of the theories of vector bin packing, we can better understand the computational complexity of our problem. We then extend their heuristics to handle practical concerns (e.g., affinity-aware placement constraints) and add support for complementary objectives (e.g., maximizing the number of deployed LRAs and minimizing the resource fragmentation).

Interference-aware scheduling. Several cluster schedulers detect interference between collocated workloads and generate schedules that avoid problematic collocations. Harmony [9] leverages a deep learning-based method to learn placement decisions in a manner that minimizes interference and maximizes performance. Bubble-Up [32] and Bubble-flux [43] throttle low priority jobs in favor of LRAs. Quasar [13] reduces interference by collocating workloads that do not compete for shared resources. Heracles [29], Elfen [44] and PerfIso [23] use isolation mechanisms to mitigate interference between collocated workloads. All these works are complementary to this work, and we leave implementation of these techniques for future work.

6. CONCLUSION

In this paper, we present LraSched, a cluster scheduler that efficiently schedules containerized LRAs in shared clusters. LraSched consists of two key components: (i) leveraging an online FM model to recommend both size and affinity of containers within an LRA automatically and (ii) employing a two-phase heuristic to efficiently schedule LRA containers along multi-dimensional resources and affinity-aware placement constraints. We have implemented a prototype of LraSched and evaluate it with testbed and simulation systems. Experimental results show that both cluster utilization and LRA performance clearly improve compared to other alternative scheduling schemes. In the future, we will further extend LraSched to deal with the resource demand dynamics during a LRA's runtime and enable it to support for batch job scheduling.

FUNDING

This work is supported by the National Science Foundation for Young Scientists of China (no. 12001328) and the Shandong Provincial Natural Science Foundation (no. ZR2019QF014 and no. ZR2020MF034).

ACKNOWLEDGMENTS

The authors are grateful to the reviewers for their valuable comments that have improved the quality of this paper.

DATA AVAILABILITY

The data underlying this article will be shared on reasonable request to the corresponding author.

REFERENCES

[1] Alibaba cluster trace. https://github.com/alibaba/clusterdata (accessed December 2020).

[2] Apache slider. http://incubator.apache.org/projects/slider.html (accessed December 2020).

[3] Docker. https://www.docker.com (accessed December 2020).

[4] Kubernetes. https://kubernetes.io (accessed December 2020).

[5] APACHE. Simplified and first-class support for services in yarn. https://issues.apache.org/jira/browse/YAR-N-4692 (accessed December 2020).

[6] Augustine, J., Banerjee, S. and Irani, S. (2006) Strip Packing with Precedence Constraints and Strip Packing with Release Times. In *Proc. of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Cambridge, Massachusetts, USA, July 30–August 2, pp. 180–189. ACM, New York.

[7] Azar, Y., Cohen, I.R. and Gamzu, I. (2013) The Loss of Serving in the Dark. In *Proc. of the Symposium on Theory of Computing Conference (STOC)*, Palo Alto, CA, USA, June 1–4, pp. 951–960. ACM, New York.

[8] Bansal, N. and Khan, A. (2014) Improved Approximation Algorithm for Two-Dimensional Bin Packing. In *Proc. of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Portland, Oregon, USA, January 5–7, pp. 13–25. SIAM, Philadelphia, USA.

[9] Bao, Y., Peng, Y. and Wu, C. (2019) Deep Learning-Based Job Placement in Distributed Machine Learning Clusters. In *Proc. of the 38th IEEE Conf. on Computer Communications (INFOCOM)*, Paris, France, April 29–May 2, pp. 505–513. IEEE Press, Piscataway, NJ, USA.

[10] Blagodurov, S., Fedorova, A., Vinnik, E., Dwyer, T. and Hermenier, F. (2015) Multi-objective Job Placement in Clusters. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Austin, TX, USA, November 15–20, pp. 1–12. ACM, New York.

[11] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010) Benchmarking Cloud Serving Systems with YCSB. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)*, Indianapolis, Indiana, USA, June 10–11, 2010, pp. 143–154. ACM, New York.

[12] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M. and Bianchini, R. (2017) Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 28–31, pp. 153–167. ACM, New York.

[13] Delimitrou, C. and Kozyrakis, C. (2014) Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proc. of the 19th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, USA, March 1–5, pp. 127–144. ACM, New York.

[14] Duchi, J.C. and Singer, Y. (2009) Efficient online and batch learning using forward backward splitting. *J. Mach. Learn. Res.*, 10, 2899–2934.

[15] Gabay, M. and Zaourar, S. Vector bin packing with heterogeneous bins: application to the machine reassignment problem. *Ann. Oper. Res.*, 242(1): 161–194, 2016.

[16] Garefalakis, P., Karanasos, K., Pietzuch, P.R., Suresh, A. and Rao, S. (2018) Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proc. of the 13th European Conf. on Computer Systems (EuroSys)*, Porto, Portugal, April 23–26, pp. 1–13. ACM, New York.

[17] Gog, I., Schwarzkopf, M., Gleave, A., Watson, R.N.M. and Hand, S. (2016) Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, USA, November 2–4, pp. 99–115. USENIX Association, Berkeley, USA.%

[18] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S. and Akella, A. (2014) Multi-resource Packing for Cluster Schedulers. In *Proc. of the Conf. of the ACM Special Interest Group on Data Communication (SIGCOMM)*, Chicago, IL, USA, August 17–22, pp. 455–466. ACM, New York.

[19] Han, R., Huang, S., Tang, F., Chang, F.-G. and Zhan, J. (2016) Accuracytrader: Accuracy-Aware Approximate Processing for Low Tail Latency and High Result Accuracy in Cloud Online Services. In *Proc. of the 45th Int. Conf. on Parallel Processing*

(ICPP), Philadelphia, PA, USA, August 16–19, pp. 278–287. IEEE Computer Society, Washington, DC.

[20] Han, Z., Tan, H., Jiang, S.H.-C., Fu, X., Cao, W. and Lau, F.C.M. (2020) Scheduling Placement-Sensitive BSP Jobs with Inaccurate Execution Time Estimation. In *Proc. of the 39th IEEE Conf. on Computer Communications (INFOCOM)*, Toronto, ON, Canada, July 6–9, pp. 1053–1062. IEEE Press, Piscataway, NJ, USA.

[21] Harren, R. and Kern, W. Improved lower bound for online strip packing. *Theory Comput. Syst*., 56(1): 41–72, 2015.

[22] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., and Stoica, I. (2011) Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA, March 30–April 1. USENIX Association, Berkeley, USA.

[23] Iorgulescu, C. et al. (2018) Perfiso: Performance Isolation for Commercial Latency-Sensitive Services. In *Proc. of USENIX Annual Technical Conf. (ATC)*, Boston, MA, USA, July 11–13, pp. 519–532. USENIX Association, Berkeley, USA.

[24] Jyothi, S.A. et al. (2016) Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, USA, November 2–4, pp. 117–134. USENIX Association, Berkeley, USA.

[25] Kitazawa, T. (2016) Incremental factorization machines for persistently cold-starting online item recommendation. CoRR, abs/1607.02858.

[26] Li, M., Liu, Z., Smola, A.J., and Wang, Y.-X. (2016) Difacto: Distributed Factorization Machines. In *Proc. of the 9th ACM Int. Conf. on Web Search and Data Mining (WSDM)*, San Francisco, CA, USA, February 22–25, pp. 377–386. ACM, New York.

[27] Lin, X., Zhang, W., Zhang, M., Zhu, W., Pei, J., Zhao, P. and Huang, J. (2018) Online Compact Convexified Factorization Machine. In *Proc. of the 2018 World Wide Web Conf. on World Wide Web (WWW)*, Lyon, France, April 23–27, pp. 1633–1642. ACM, New York.

[28] Liu, L. and Xu, H. (2018) Elasecutor: Elastic executor scheduling in data analytics systems. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)*, Carlsbad, CA, USA, October 11–13, pp. 107–120. ACM, New York.

[29] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. (2016) Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst.*, 34(2): 6:1–6:33.

[30] Lu, C., Ye, K., Xu, G., Xu, C.-Z., and Bai, T. (2017) Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *Proc. of IEEE Int. Conf. on Big Data*, Boston, MA, USA, December 11–14, pp. 2884–2892. IEEE Computer Society, Washington, DC.

[31] Mao, H., Schwarzkopf, M., Venkatakrishnan, S.B., Meng, Z. and Alizadeh, M. (2019) Learning scheduling algorithms for data processing clusters. In *Proc. of the ACM Special Interest Group on Data Communication (SIGCOMM)*, Beijing, China, August 19-23, pp. 270–288. ACM, New York.

[32] Mars, J., Tang, L., Hundt, R., Skadron, K. and Bubble-up, M.L.S. (2011) Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proc. of the 44rd Annual IEEE/ACM Int. Symposium on Microarchitecture (MICRO)*, Porto Alegre, Brazil, December 3–7, pp. 248–259. ACM, New York.

[33] McMahan, H.B. (2011) Follow-the-Regularized-Leader and Mirror Descent: Equivalence Theorems and L1 Regularization. In *Proc. of the 14th Int. Conf. on Artificial Intelligence and Statistics (AISTATS)*, Fort Lauderdale, USA, April 11–13, pp. 525–533. JMLR, Brookline, USA.

[34] MSR-TR-2011-9 (2011) Validating heuristics for virtual machines consolidation. Microsoft Research, California, USA.

[35] Panigrahy, R., Talwar, K. and Uyeda, L. and Udi Wieder. Heuristics for vector bin packing. https://www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/ (accessed December 2020).

[36] Park, J.W., Tumanov, A., Jiang, A.H., Kozuch, M.A. and Ganger, G.R. (2018) 3Sigma: Distribution-Based Cluster Scheduling for Runtime Uncertainty. In *Proc. of the 13th European Conf. on Computer Systems (EuroSys)*, Porto, Portugal, April 23–26, pp. 1–17. ACM, New York.

[37] Rendle, S. (2010) Factorization Machines. In *Proc. of the 10th IEEE Int. Conf. on Data Mining (ICDM)*, Sydney, Australia, December 14–17, pp. 995–1000. IEEE Computer Society, Washington, DC.

[38] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M. and Omega, J.W. (2013) Flexible, Scalable Schedulers for Large Compute Clusters. In *Proc. of the 8th ACM European Conf. on Computer Systems (EuroSys)*, Prague, Czech Republic, April 14–17, pp. 351–364. ACM, New York.

[39] Stillwell, M., Schanzenbach, D., Vivien, F., and Casanova, H. (2010) Resource allocation algorithms for virtualized service hosting platforms. *J. Parallel Distributed Comput.*, 70(9): 962–974.

[40] Vavilapalli, V.K. et al. (2013) Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)*, Santa Clara, CA, USA, October 1–3, pp. 5:1–5:16. ACM, New York.

[41] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. and Wilkes, J. (2015) Large-Scale Cluster Management at Google with Borg. In *Proc. of the 10th European Conf. on Computer Systems (EuroSys)*, Bordeaux, France, April 21–24, pp. 1–17. ACM, New York.

[42] Xiao, L. (2010) Dual averaging methods for regularized stochastic learning and online optimization. *J. Mach. Learn. Res.*, 11, 2543–2596.

[43] Yang, H., Breslow, A.D., Mars, J., and Tang, L.. (2013) Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proc. of the 40th Annual Int. Symposium on Computer Architecture (ISCA)*, Tel-Aviv, Israel, June 23–27, pp. 607–618. ACM, New York.

[44] Xi, Y., Blackburn, S.M. and scheduling, K.S.M.K.E. (2016) Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *Proc. of USENIX Annual Technical Conf. (ATC), Denver, CO, USA, June 22–24*, pp. 309–322. USENIX Association, Berkeley, USA.

[45] Zhang, Q., Zhou, R., Chuan, W., Jiao, L. and Li, Z. (2020) Online Scheduling of Heterogeneous Distributed Machine Learning Jobs. In *Proc. of the 21th ACM Int. Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (Mobihoc)*, Virtual Event, USA, October 11–14, pp. 111–120. ACM, New York.