Tetris: Scheduling Long-Running Workloads for Load Balancing in Shared Containerized Clusters

Xiyue Shen[a], Fei Xu[a,*], Yifan Liu[b], Tao Song[b], Shuohao Lin[a], Li Chen[c], Fen Xiao[d]

[a] *Shanghai Key Laboratory of Multidimensional Information Processing, School of Computer Science and Technology, East China Normal University*
[b] *Shanghai F7 Networks Technology Inc.*
[c] *University of Louisiana at Lafayette*
[d] *Tencent Inc.*

Abstract

Long-running containerized workloads (*e.g.,* online cloud services, machine learning) are increasingly prevailing in shared production clusters. As the request loads of such long-running containers typically show *time-varying* patterns, container scheduling is essential to improving the cluster resource utilization and workload performance. Existing cluster schedulers mainly focus on optimizing either the *short-term* benefits of cluster load balancing or the *initial placement* of long-running containers on servers. Such schedulers, however, would inevitably bring a noticeable number of *invalid migrations* (*i.e.,* containers are migrated back and forth between two servers over a short time window), leading to serious SLO violations. In this paper, we propose *Tetris*, a *model predictive control* (MPC)-based container scheduling strategy to judiciously migrate long-running workloads for cluster load balancing.

*The corresponding author is Fei Xu.

Email addresses: `fxu@cs.ecnu.edu.cn` (Fei Xu), `yangjl@f7sys.com.cn` (Tao Song), `li.chen@louisiana.edu` (Li Chen), `cherryxiao@tencent.com` (Fen Xiao)

Specifically, we first build a discrete-time dynamic model to formulate a *long-term* optimization problem of container scheduling. *Tetris* then solves such an optimization problem using two key components: (1) a container resource predictor, which leverages time-series analysis methods to accurately predict the container resource consumption; (2) an MPC-based container scheduler that jointly optimizes the load balancing and migration cost of long-running containers *over a certain sliding time window*. We implement a prototype of *Tetris* based on K8s, and evaluate *Tetris* with prototype experiments on Amazon EC2 and large-scale simulations driven by Alibaba cluster trace v2018. Experiment results show that *Tetris* can improve the cluster load balancing degree by up to 77.8% without incurring any service level objective (SLO) violations, in comparison to the state-of-the-art container scheduling strategies, yet with acceptable runtime overhead.

Keywords:
long-running containerized workloads, load balancing, migration cost, container scheduling

1. Introduction

Large production clusters are commonly hosting various *long-running* containerized workloads, ranging from online Web services to streaming processing systems and machine learning applications [1]. Unlike traditional batched jobs that are typically executed within seconds to minutes, these long-running containerized workloads generally last for hours to months [2], and often have stringent service level objectives (SLOs) [3]. However, as the request loads of long-running workloads are *time-varying* [4], sudden and unpredictable changes of container resource consumption can cause severe contention of shared cluster resources. Such cluster load imbalance can result in many potential service level objective (SLO) violations to workloads. Consequently, the mainstream cluster schedulers such as Borg [5] and Kubernetes [6] enable container scheduling to achieve load balancing [7] in shared production clusters.

Though the existing container scheduling policies such as Sandpiper [7] and Medea [2] perform well in achieving cluster load balancing, there still exist a noticeable number of *invalid migrations* (*i.e.,* containers are migrated back and forth between two servers over a short time window) in shared production clusters. As evidenced by our motivational analysis (in Sec. 2.2) on Alibaba cluster trace v2018 [8], the number of invalid migrations is over $1,200$, accounting for more than 50% of container migrations within the window size of three timeslots. Such invalid migrations are likely to make the workloads hosted on the migrated containers suffer from serious SLO violations, leading to unexpected performance interference to the containers that are *co-located* on servers (*i.e.,* migration cost). Our motivation experiments in Sec. 2.2 also reveal that the invalid migrations can significantly reduce the number of requests processed by an Apache Tomcat Web server hosted on a migrated container by up to 99.8%. Accordingly, it is critical for the cluster scheduler to avoid invalid migrations during the process of container scheduling, especially for long-running workloads.

Unfortunately, many research efforts have been devoted to making *short-term* scheduling decisions based on the *current* cluster status for workload consolidation [9] or load balancing [10]. Though such scheduling policies can achieve short-term (*e.g.,* the current or upcoming timeslot) benefits, they are oblivious to the *time-varying* resource consumption of long-running workloads, which is actually the *root cause of invalid migrations* as discussed in Sec. 2.2. There have also been recent works on the *long-term* optimization of container scheduling, which are reinforcement learning (RL)-based [1, 11] or control-based [12, 13] scheduling policies, to *partially* tackle the issue of invalid migrations. Nevertheless, these techniques solely focus on optimizing the *initial placement* or resource auto-scaling of containerized workloads (*i.e., where to migrate*) over the entire time period (*i.e., infinite future*). The containers to be scheduled (*i.e., which to migrate*) and the *migration cost* of containers have surprisingly received little attention. Accordingly, such *long-term* optimization techniques can still cause unexpected SLO violations, as evidenced in Sec. 2.3. As a result, there has been a paucity of research attention paid to developing container scheduling policies to *fully* deal with the invalid migrations of long-running workloads in containerized clusters.

To fill this gap, we propose *Tetris*, a model predictive control [14] (MPC)-based container scheduling strategy to judiciously make migration decisions for long-running containerized workloads. *Tetris* simply optimizes container scheduling *over a certain sliding time window* rather than *the infinite future* (as in RL-based method [3]) for two reasons: *first*, the prediction accuracy of container resource consumption dramatically decreases as the prediction window size increases. Our prediction results using time-series techniques (in Sec. 5.2) indicate that the prediction error can exceed 20% as the prediction window size reaches 6; *second*, blindly increasing the time window size can significantly increase the number of samples of *Tetris*, which requires noticeable computation overhead to obtain container scheduling plans. To the best of our knowledge, *Tetris* is the first attempt to achieve the *long-term* optimization of container scheduling to *circumvent invalid migrations as many as possible*, by jointly optimizing the cluster load balancing and container migration cost over a certain sliding time window. Our main contributions are summarized as follows.

▷ *First,* we build a discrete-time dynamic model for shared containerized clusters to capture the time-varying resource consumption of containers and the corresponding mapping of containers on servers (Sec. 3). Based on such a model, we further devise a cost function of container scheduling *over a time window* and formulate our *long-term* workload scheduling optimization problem based on MPC, by jointly considering the cluster load imbalance degree and migration cost of containers.

▷ *Second,* we design an MPC-based container scheduling strategy to achieve *long-term* scheduling optimization for cluster load balancing (Sec. 4). Specifically, *Tetris* first accurately predicts the container resource consumption within a time window using time-series analysis methods. It then leverages the Monte Carlo method [15] to judiciously identify the container scheduling decision for each timeslot, by minimizing our formulated cost function of container scheduling. In particular, we calculate the upper and lower bounds of server load imbalance degree to classify the migration source and destination servers, so that the complexity of *Tetris* container scheduling strategy can be significantly reduced.

▷ *Finally,* we implement a prototype of *Tetris*[1] based on K8s. We eval-

[1] https://github.com/icloud-ecnu/tetris

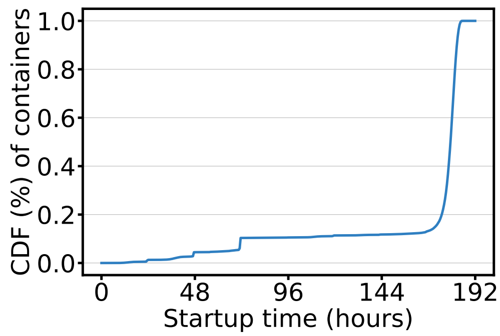


Figure 1: Container uptime in Alibaba cluster trace v2018.

uate the effectiveness and runtime overhead of *Tetris* with prototype experiments on 10 EC2 instances (*i.e.,* 60 containers) and large-scale simulations driven by Alibaba cluster trace v2018 (Sec. 5). Compared with the conventional scheduling strategy (*i.e.,* Sandpiper [7]) and the state-of-the-art RL-based method (*i.e.,* Metis^{+}, a modified version of Metis [3]), *Tetris* is able to improve the cluster load balancing degree by up to 77.8% while cutting down the migration cost by up to 79.5%, yet with acceptable runtime overhead.

2. Background and Motivation

2.1. Long-running Containerized Workloads

Large-scale shared production clusters often host a number of long-running containers in response to latency-critical user requests [2]. A variety of long-running workloads including streaming, machine learning, and Web applications run on them, in addition to storage services generated by multi-tenant NoSQL databases, *etc.* [16]. Taking Alibaba as an example, its online services are mainly user-interactive and long-running applications that run on containers, such as search engines and online shopping [17]. As depicted in Fig. 1, over 80% of containers in the Alibaba production cluster last for more than 175 hours[2]. Due to the unpredictability of user requests, such long-running containerized workloads have diverse demands on hardware resources including CPU, memory, network and disk I/O, which are generally dynamic variability and uncertainty in the amount of resource requests [18]. Furthermore, the co-location of containers can lead to contention for shared resources, which can easily cause performance interference and resource wastage in the cluster.

Based on the above, enabling adequate scheduling of long-running containerized workloads is essential to cluster load balancing. In shared production clusters, K8s achieves load balancing among servers by carrying out container migrations [6]. During the migration process, the container needs to stop services first, and then it is terminated on the source server and later restarted on the destination server, and finally it replicates the memory/storage data to restore services. Accordingly, the container scheduling inevitably brings a noticeable amount of migration cost, which requires to be considered during the scheduling of containers.

2.2. Invalid Migrations of Long-Running Workloads

Load balancing is an effective way to alleviate resource contention in shared containerized clusters [7]. While many existing scheduling policies perform well in load balancing, they usually focus on the *short-term* benefits of container scheduling. As discussed above, the resource consumption of long-running containers are commonly *time-varying* (*i.e.,* fluctuating wildly

[2] The total time in the Alibaba cluster trace v2018 is 192 hours.

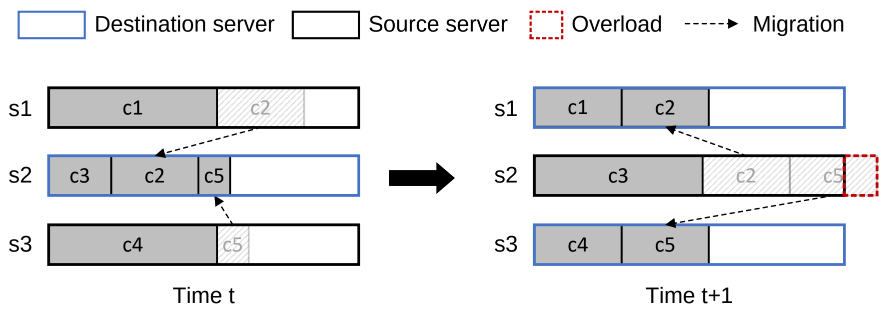


Figure 2: Illustration of invalid migration. Containers $c2$ and $c5$ are migrated back and forth among servers $s1$, $s2$, and $s3$.

over time). Such a *short-term* optimization of container scheduling policy can make *invalid migration* decisions for containers, resulting in unpredictable migration cost and potential SLO violations.

Specifically, we take a cluster of three servers hosting five containers illustrated in Fig. 2 as an example. The cluster scheduler first migrates container $c2$ and $c5$ from server $s1$ and $s3$, respectively, to server $s2$, in order to obtain a *temporary* benefits of load balancing at time t. Unfortunately, server $s2$ becomes overloaded as the resource consumption of three containers (*i.e.,* $c3$, $c2$, $c5$) dramatically increases, which triggers two container migrations (*i.e.,* container $c2$, $c5$) from server $s2$ to $s1$ and $s3$, respectively. In such a case, naive scheduling policies are likely to make long-running containers migrated back and forth among servers, thereby causing heavy and unnecessary migration cost to containers. Accordingly, we formally define *invalid migrations* as in Definition 1.

Definition 1. *If a container is migrated back and forth between two servers over a short time window* $[t, t+w]$*, we define such container migrations as invalid migrations within a time window size* w*.*

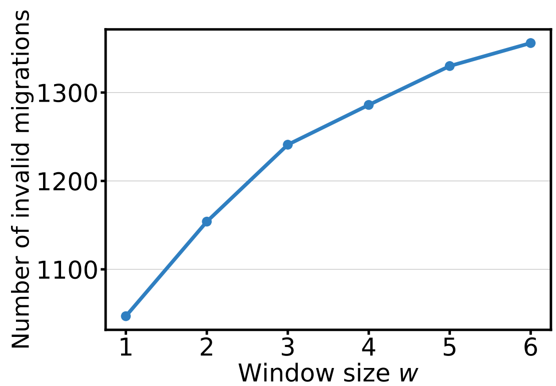


Figure 3: Number of invalid migrations within a time window size varying from 1 to 6.

Moreover, we validate the prevalence of invalid migrations using the Alibaba cluster trace v2018 with an 8-day period [8]. As shown in Fig. 3, we observe that the number of invalid migrations increases as the scheduling time window increases. Specifically, the cumulative number of invalid migrations exceeds $1,200$ (*i.e.,* around 50% of container migrations) as w is set as 3, and the number reaches around $1,400$, accounting for 75.3% of container migrations when w is set as 6. To illustrate the performance impact of invalid migrations, we conduct experiments on a cluster of 10 servers (*i.e.,* 60 containers) deployed with Apache Tomcat Web server. Our experiment results reveal that 61.7% of the containers are affected by invalid migrations. In addition, the number of requests processed by a migrated container can be reduced by up to 99.8% (74.0% on average), severely degrading the quality of long-running Web service. Therefore, it is essential to design a *long-term* container scheduling strategy to alleviate invalid migrations by explicitly considering the future resource consumption of containers.

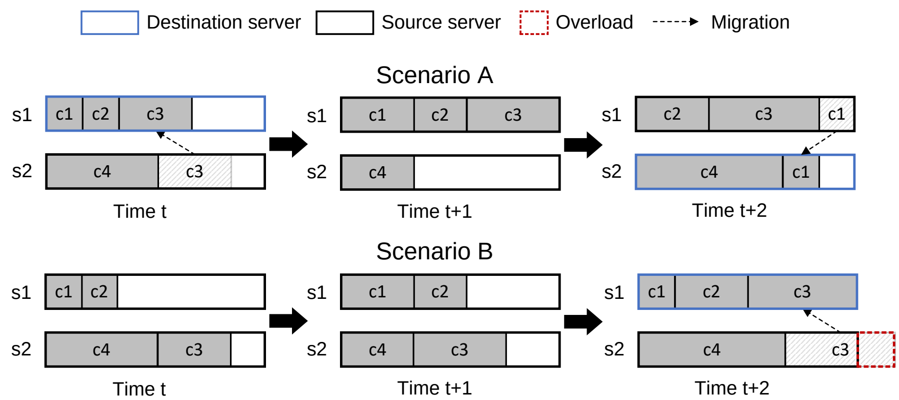


Figure 4: An illustrative example of MPC-based container scheduling (our proposed *Tetris* in scenario A), compared with RL-based container scheduling (Metis$^+$ [3] in scenario B).

2.3. An Illustrative Example

To avoid invalid migrations, we design *Tetris* in Sec. 4, a *simple yet effective* container scheduling strategy that leverages the MPC approach to judiciously migrate long-running workloads for achieving the cluster load balancing. In particular, MPC adopts a compromise strategy which allows the current timeslot to be optimized while taking *finite future timeslots* in account [19]. To illustrate how *Tetris* works, we show a motivation example in Fig. 4 by comparing *Tetris* with a RL-based scheduling method (*i.e.*, Metis$^+$ in Scenario B, a modified version of Metis [3]). Though Metis$^+$ greedily achieves the optimal load balancing degree and minimizes the migration cost over the *entire* time period of $[t, t+2]$, it can still overload server $s2$ at time $t+2$ and trigger the migration of container $c3$ from server $s2$ to $s1$. In contrast, our MPC approach (*i.e.*, *Tetris* in Scenario A) jointly optimizes the load balancing degree and migration cost for two *sliding* timeslots (*i.e.*, $[t, t+1]$, and $[t+1, t+2]$) and guarantees SLOs of all containerized workloads. Accordingly, the RL-based method can optimize the scheduling objective *over the entire time period* at the cost of overloading a certain number of containers, while *Tetris* achieves the *long-term* optimization for container scheduling *within a certain sliding time window*. In addition, the RL-based method has the following problems such as requiring repeated training on a large amount of high-quality data samples and lacking interpretability as well as poor scalability [3], which will be further validated in Sec. 5.4.

Summary: Avoiding invalid migrations is critical to container scheduling, and such invalid migrations are mainly caused by the *short-term* (*e.g.,* the current or upcoming timeslot) optimization of container scheduling. Meanwhile, greedily optimizing container scheduling over the entire time period (*i.e., infinite future*) are likely to cause unexpected SLO violations. Accordingly, there is a compelling need to design a *long-term* container scheduling strategy, by jointly optimizing the cluster load balancing and migration cost of containers within a certain sliding time window.

3. Model and Problem Formulation

In this section, we first build a discrete-time dynamic model to capture the load imbalance degree of clusters and migration cost of containers. Next, we formulate a container scheduling optimization problem based on our dynamic model. The key notations in our model are summarized in Table 1.

3.1. Discrete-Time Dynamic Model

We consider a containerized cluster including a set of servers $\mathcal{M}$ hosting a set of containers $\mathcal{N}$. We assume the time t is discrete and slotted, where $t = t_0, t_1, ..., t_W$ and W is the time window size. For each container $k \in \mathcal{N}$, its CPU and memory resource consumption at each time t is recorded as $cpu^k(t)$

Table 1: Key notations in our discrete-time dynamic model.

Notation	Definition
$\mathcal{M}$, $\mathcal{N}$	Sets of servers and containers
W	Time window size
$C_b(t)$, $C_m(t)$	Cluster load imbalance degree and migration cost at time t
$C_{mig}^k(t)$	Migration cost of container k at time t
$cpu^k(t)$	CPU consumption of container k at time t
$mem^k(t)$	Memory consumption of container k at time t
$\overline{CPU(t)}$	Average CPU consumption of a server in the cluster at time t
$\overline{MEM(t)}$	Average memory consumption of a server in the cluster at time t
$CPU_i^{cap}(t)$	Available capacity of CPU resource of server i at time t
$MEM_i^{cap}(t)$	Available capacity of memory resource of server i at time t
α	Normalized parameter of migration cost to load imbalance degree
β	Normalized parameter of memory resources to CPU resources
$x_i^k(t)$	Indicator whether container k is on server i at time t
$m^k(t)$	Indicator whether container k is migrated at time t

and $mem^k(t)$, respectively. For each server $i \in \mathcal{M}$, $CPU_i(t)$ and $MEM_i(t)$ represent its total CPU and total memory resource consumption at time t, respectively.

Modeling cluster load imbalance degree at each timeslot. We use the variance of resource consumption of all cluster servers to represent the load imbalance degree of the cluster. A larger degree value indicates a severer load imbalance situation. By taking CPU and memory resources as an example, the cluster load imbalance degree is represented by calculating the weighted sum of the load imbalance degrees of each resource of servers, which is given by

$$C_b(t) = \frac{1}{|\mathcal{M}|-1} \sum_{i\in\mathcal{M}} \Bigg(\Big(\sum_{k\in\mathcal{N}} x_i^k(t) \cdot cpu^k(t) - \overline{CPU(t)} \Big)^2 + \beta \cdot \Big(\sum_{k\in\mathcal{N}} x_i^k(t) \cdot mem^k(t) - \overline{MEM(t)} \Big)^2 \Bigg), \tag{1}$$

where $\beta \in [0,1]$ denotes the normalized parameter of memory resources to CPU resources. In practice, the value of β can be empirically determined as the square of the ratio of CPU to memory resource consumption of workloads. Also, $x_i^k(t)$ denotes whether the container k is hosted on the server i.

$$x_i^k(t) = \begin{cases} 1, & \text{if container } k \text{ runs on server } i, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Then, we proceed to denote the average CPU resource consumption $\overline{CPU(t)}$ and average memory resource consumption $\overline{MEM(t)}$ of a server in the cluster as below,

$$\overline{CPU(t)} = \frac{1}{|\mathcal{M}|} \sum_{k\in\mathcal{N}} cpu^k(t), \tag{3}$$

$$\overline{MEM(t)} = \frac{1}{|\mathcal{M}|} \sum_{k\in\mathcal{N}} mem^k(t). \tag{4}$$

Modeling migration cost at each timeslot. We use the sum of unit cost of the migrated containers to denote the migration cost. A larger number of migrated containers and a larger unit cost of each migrated container indicate a severer migration cost. Accordingly, the migration cost is formulated as

$$C_m(t) = \sum_{k\in\mathcal{N}} C_{mig}^k(t)\cdot m^k(t), \tag{5}$$

where $m^k(t)$ denotes whether container k is migrated at time t, which is given by Eq. (6). $C_{mig}^k(t)$ is the unit migration cost of container k. As elaborated in Sec. 2.1, K8s [6] implements the container migration with three steps: service stop, container deletion and restart, and data replication. In particular, the data replication overhead is proportional to the container's memory footprint [20], while the overhead of the other two steps can be considered as a constant value δ. Hence, the unit migration cost of a container can be given by

$$\begin{aligned} m^k(t) &= \sum_{i\in\mathcal{M}} x_i^k(t)\cdot\big(1 - x_i^k(t-1)\big), & (6)\\ C_{mig}^k(t) &= \delta + \gamma\cdot mem^k(t), & (7)\end{aligned}$$

where γ is a model coefficient obtained by workload profiling, and $mem^k(t)$ is the memory footprint of container k at time t.

To sum up, we further formulate the *cost function $C(t)$ of container scheduling at each timeslot*. Based on the formulations above, we further combine the cluster load imbalance degree and the migration cost as below,

$$C(t) = C_b(t) + \alpha\cdot C_m(t), \tag{8}$$

where $\alpha \in [0,1]$ denotes the normalized parameter of migration cost to cluster load imbalance degree. In more detail, we substitute Eq. (3) and Eq. (4) into Eq. (1) and substitute Eq. (6) and Eq. (7) into Eq. (5), yielding Eq. (9) and

Eq. (10), respectively, which are calculated as

$$C_b(t) = \frac{2}{|\mathcal{M}|-1} \sum_{i\in\mathcal{M}} \sum_{k,l\in\mathcal{N}} x_i^k(t) x_i^l(t) \cdot \big(cpu^k(t) cpu^l(t) + \beta \cdot mem^k(t) mem^l(t)\big) + C_1, \tag{9}$$

$$C_m(t) = C_2 - \sum_{k\in\mathcal{N}} \sum_{i\in\mathcal{M}} \big(\delta + \gamma \cdot mem^k(t)\big) \cdot x_i^k(t) x_i^k(t-1), \tag{10}$$

where $C_1 = -\frac{|\mathcal{M}|\cdot\left(\overline{CPU(t)}^2+\beta\cdot\overline{MEM(t)}^2\right)+\sum_{k\in\mathcal{N}}\left(cpu^k(t)^2+\beta\cdot mem^k(t)^2\right)}{|\mathcal{M}|-1}$ and $C_2 = \delta \cdot |\mathcal{N}| + \gamma \cdot \sum_{k\in\mathcal{N}} mem^k(t)$ are both constant values. The detailed mathematical derivations can be found in Appendix .1. By analyzing Eq. (9), the cluster load imbalance degree is determined by the sum of the pairwise multiplications between the resource consumption of containers hosted on each server. By analyzing Eq. (10), the migration cost across the cluster is determined by the negative of the sum of migration costs of all migrated containers.

3.2. Workload Scheduling Optimization Problem over a Time Window

Based on our discrete-time dynamic model above, we proceed to formulate the *long-term* optimization problem of container scheduling based on MPC. At each time t, MPC leverages the predicted container resource consumption (*i.e.*, $cpu^k(t)$, $mem^k(t)$) to make scheduling decisions (*i.e.*, judiciously deciding $x_i^k(t)$) to minimize the cost function $C(t)$ over the time window (as in Eq. (11)). We assume the scheduling starts at time t_1, and our optimization problem can be formulated as

$$\min_{x_i^k(t)} \quad \sum_{t=t_1}^{t_W} C(t) \tag{11}$$

$$\text{s.t.} \quad \sum_{i\in\mathcal{M}} x_i^k(t) = 1, \quad \forall k \in \mathcal{N}, t \in [t_1, t_W] \tag{12}$$

$$\sum_{k\in\mathcal{N}} x_i^k(t) \cdot cpu^k(t) \le CPU_i^{cap}(t), \quad \forall i \in \mathcal{M}, t \in [t_1, t_W] \tag{13}$$

$$\sum_{k\in\mathcal{N}} x_i^k(t) \cdot mem^k(t) \le MEM_i^{cap}(t). \quad \forall i \in \mathcal{M}, t \in [t_1, t_W] \tag{14}$$

The hard constraints in our optimization problem are described as follows: Constraint (12) limits each container to be hosted only on one server per time to avoid scheduling conflicts. Constraints (13)-(14) ensure the total resource consumption on each server cannot exceed the server resource capacity.

Problem Analysis. For each timeslot, the minimization problem defined in Eq. (11) can be easily reduced to a *multiprocessor scheduling problem (MSP)* [21]. MSP can be considered as finding a schedule for multiple tasks to be executed on a multiprocessor system at different execution time, so that the completion time can be minimized. Such a scheduling problem is known to be NP-hard [22]. Moreover, Eq. (8) is a multi-objective optimization problem that jointly considers minimizing the migration cost and the cluster load imbalance degree, which further makes such a problem hard to solve. As a result, our optimization problem of Eq. (11) turns out to be NP-hard. In the upcoming section, we turn to leveraging the Monte Carlo method [15] to solve our long-term workload scheduling optimization problem.

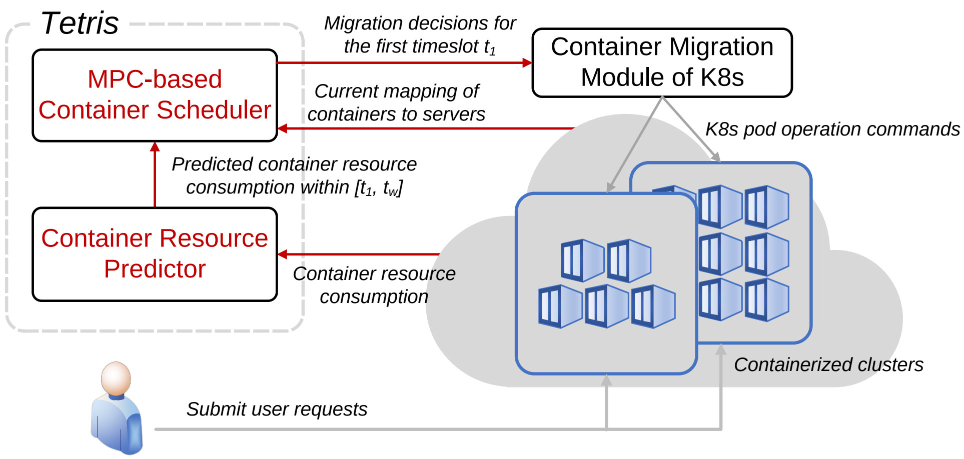


Figure 5: Overview of *Tetris* prototype in containerized clusters.

4. Tetris Design

Based on our discrete-time dynamic model and optimization problem defined in Sec. 3, we proceed to design *Tetris*, a simple yet effective container scheduling strategy that leverages the MPC approach to jointly optimize the cluster load balancing degree and container migration cost for a certain time window.

4.1. Overview of Tetris

As illustrated in Fig. 5, *Tetris* comprises two pieces of modules including a *container resource predictor* (Sec. 4.2) and a *MPC-based container scheduler* (Sec. 4.3). After users submit workload requests to the containerized cluster, *Tetris* first leverages the *predictor* to estimate the resource consumption of containers over a given time window W, which is input to the *scheduler* for calculating the scheduling cost of containers. By *jointly* optimizing the cluster load balancing degree and container migration cost, the *scheduler* further decides the appropriate migration plans during the period of time window W,

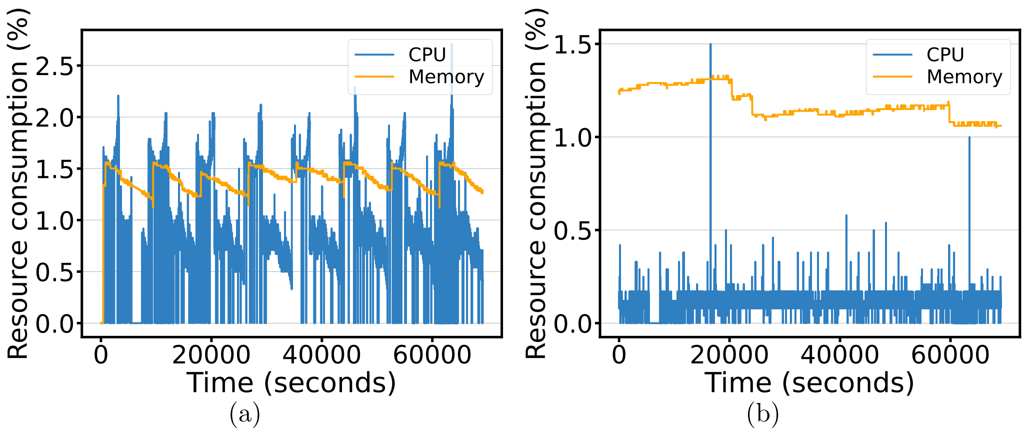


Figure 6: Resource consumption of containers over time: (a) a periodic container ($id = 71,417$), and (b) an aperiodic container ($id = 58$) in Alibaba cluster trace v2018.

which will be elaborated in Alg. 1. Finally, we implement a *container migration module of K8s*, which performs the appropriate migration decisions for the *first* timeslot in the containerized cluster, while discarding the migration decisions for the remaining timeslots. In particular, such a migration module can convert container scheduling decisions into a series of K8s pod[3] operations (*i.e.*, pod deletion and creation commands executed on migration source and destination servers). The prototype of *Tetris* is implemented based on K8s, with over $1,000$ lines of Python and Linux Shell codes which are publicly available on GitHub (`https://github.com/icloud-ecnu/tetris`).

4.2. Predicting Container Resource Consumption

We first classify workloads as *periodic* and *aperiodic* containers shown in Fig. 6, as most containerized workloads show a diurnal pattern [17]. Specifically, we first obtain the *frequency* of container resource consumption values

[3]Note that each pod in our K8s cluster hosts one container in our experimental setup.

and calculate its corresponding *periodicity*. With such a periodicity value, we then slice the container resource consumption into a number of segments. We finally calculate the Pearson correlation coefficients between every two consecutive segments and decide whether such coefficients exceed a given periodicity threshold thr_{period} (*e.g.,* 0.85). If the coefficients exceed the threshold, we consider such a container as *periodic*. Otherwise, we consider the container as *aperiodic*.

Next, we proceed to predict the resource consumption for such two types of containers separately. In more detail, we first adopt ARIMA [23] to predict the resource consumption of *periodic* containers. We use the *historical* container resource consumption to pre-train a *shared* ARIMA model for newly-launched containers. To improve the prediction accuracy, we further leverage incremental learning [24] to train a *personalized* ARIMA model for each container that has been running for a certain period of time, and we can update the model periodically. In particular, we mainly obtain three key parameters (*i.e.,* the autoregressive process of order p, the moving average process of order q, and the difference order d) of the ARIMA model. Second, we leverage LSTM [25] to predict the resource consumption of *aperiodic* containers. We adopt a 4-layer stacked LSTM model with 50 neurons per layer to increase the depth of the neural network, and add a *Dropout* layer between every two LSTM layers to reduce overfitting. To keep the LSTM model simple and alleviate the impact of error propagation, we directly conduct the multistep-ahead predictions and the number of prediction steps is set as W.

4.3. MPC-based Container Scheduling

We leverage the MPC approach to make container scheduling decisions over the time window W. Specifically, we use a $|\mathcal{M}| \times |\mathcal{N}|$ matrix $\mathbf{X}_t$ of $x_i^k(t)$ ($i \in \mathcal{M}, k \in \mathcal{N}$) to denote the mapping of containers to servers at time $t \in [t_0, t_W]$. According to MPC, we first solve the scheduling optimization problem in Sec. 3.2 to obtain $\mathbf{X}_t, \forall t \in [t_1, t_W]$, at the current time t_0. Then, we only perform the container scheduling decision $\mathbf{X}_{t_1}$ for the first timeslot t_1. After the containers are scheduled to migration destination servers at the "current" time t_1, we continue solving the scheduling optimization problem and then perform the container scheduling decisions for the "first" timeslot t_2. To maintain load balancing of the containerized cluster, *Tetris* can be *periodically* (*e.g.,* for several minutes or hours) executed at each timeslot.

In more detail, we solve the scheduling optimization problem based on the Monte Carlo method [15] as discussed in Sec. 3.2. As shown in Alg 1, we take Z *sample* solutions in total, and each sample solution includes a set of container scheduling decisions $\mathbf{X}_t, \forall t \in [t_1, t_W]$ over the time window W (lines 1 to 2). To obtain the container scheduling decisions for each timeslot t, we design two phases in Alg. 1, *i.e.,* server classification (lines 3 to 9) and container scheduling (lines 10 to 19), which are elaborated as follows.

Phase I: Server classification. We classify the cluster servers into three categories, *i.e.,* migration source servers and destination servers as well as the remaining servers (lines 3 to 9). Based on the cluster load imbalance degree (*i.e.,* Eq. (9)) defined in our dynamic model, we can obtain the

Algorithm 1: *Tetris*: Long-term container scheduling algorithm for jointly optimizing load balancing and migration cost of containers.

Input: Current mapping $\mathbf{X}_{t_0}$ of container set $\mathcal{N}$ to server set $\mathcal{M}$, container CPU and memory resource consumption over a time window W, number of samples Z, number of trials K.

Output: Container scheduling decisions $\mathbf{X}_{t_1}^{min}$ at time t_1.

for ***all*** $z \in [1, Z]$ **do**
    for ***all*** $t \in [t_1, t_W]$ **do**
        // server classification
        Initialize: the $load_i(t)$ threshold for migration source server $load_{src}(t) \leftarrow$ Eq. (16) and that for migration destination server $load_{dest}(t) \leftarrow$ Eq. (17);
        for ***each*** *server* $i \in \mathcal{M}$ **do**
            Calculate the load imbalance degree $load_i(t) \leftarrow$ Eq. (15);
            if $load_i(t) > load_{src}(t)$ **then**
                Put server i to migration source server set $\mathcal{M}_{src}$;
            if $load_i(t) < load_{dest}(t)$ **then**
                Put server i to migration destination server set $\mathcal{M}_{dest}$;
        // container scheduling
        for ***all*** $k \in [1, K]$ **do**
            for ***each*** *server* $i \in \mathcal{M}_{src}$ **do**
                Obtain the container set to be migrated $\mathcal{N}_{mig}$;
            for ***each*** *container* $c \in \mathcal{N}_{mig}$ **do**
                for ***each*** *server* $i \in \mathcal{M}_{dest}$ **do**
                    Calculate the container scheduling cost $C(t) \leftarrow$ Eq. (8) if container c is migrated to server i;
                Update $\mathbf{X}_t$ by migrating container c to the server i with the smallest $C(t)$;
            if $\mathbf{X}_t$ *satisfies scheduling constraints (12) – (14)* **then**
                break;
            Update $\mathbf{X}_t \leftarrow \mathbf{X}_{t-1}$; // $\mathbf{X}_t$ cannot satisfy constraints
    Calculate the container scheduling cost over the time window W $cost_z \leftarrow$ Eq. (11) of $\mathbf{X}_t$ for each sample z;
Obtain $\mathbf{X}_t^{min} \leftarrow \mathbf{X}_t$ with the minimized $cost_z$ among all Z samples;
return: Scheduling decisions for the first timeslot (*i.e.*, $\mathbf{X}_{t_1}^{min}$).

simplified *load imbalance degree* (*i.e.*, $load_i(t)$) for *each server* i given by

$$load_i(t) = \sum_{k,l \in \mathcal{N}} x_i^k(t) x_i^l(t) \cdot \big(cpu^k(t) cpu^l(t) + \beta \cdot mem^k(t) mem^l(t)\big). \quad (15)$$

Obviously, the CPU and memory resource consumption of each server are $\overline{CPU(t)}$ (*i.e.*, Eq. (3)) and $\overline{MEM(t)}$ (*i.e.*, Eq. (4)), respectively, when the cluster reaches the *"ideal" load-balanced state*. As the load imbalance degree of each server $load_i(t)$ can vary with the number of hosted containers and the resource consumption of containers, we obtain the thresholds of $load_i(t)$ for migration source servers and destination servers as in Theorem 1.

Theorem 1. *Given a server hosting a set of containers with the average CPU and memory resource consumption (i.e.,* $\overline{CPU(t)}$ *and* $\overline{MEM(t)}$ *at time* t*, the thresholds of* $load_i(t)$ *for migration source server and destination server can be formulated as below:*

$$load_{src}(t) = \left(\frac{1}{2} - \frac{1}{2|\mathcal{N}|}\right) \cdot \left(\overline{CPU(t)}^2 + \beta \cdot \overline{MEM(t)}^2\right), \quad (16)$$

$$load_{dest}(t) = \frac{1}{2}\Big(\overline{CPU(t)}^2 + \beta \cdot \overline{MEM(t)}^2 - \sum_{k=1}^{n(t)} \big(cpu^k(t)^2 + \beta \cdot mem^k(t)^2\big)\Big). \quad (17)$$

By sorting containers in a descending order with the CPU and memory resource consumption, $cpu^k(t)$ *and* $mem^k(t)$ *represent the CPU and memory consumption of the* k*-th container at time* t*, respectively.* $n(t)$ *denotes the minimum number of containers which satisfies* $\sum_{k=1}^{n(t)} cpu^k(t) \leq \overline{CPU(t)}$ *and* $\sum_{k=1}^{n(t)} mem^k(t) \leq \overline{MEM(t)}$*.*

Proof. The proof can be found in Appendix .2. □

Phase II: Container scheduling. To make Alg. 1 practical, we sample a set of containers to be migrated $\mathcal{N}_{mig}$ from the set of migration source servers $\mathcal{M}_{src}$. To achieve a more comprehensive coverage of possibilities with a small number of samples, we adopt Latin Hypercube Sampling (LHS) with a sampling rate of v per migration source server (lines 11 to 12). For each container to be migrated, we further select the migration destination server with the smallest container scheduling cost and then update $\mathbf{X}_t$ (lines 13 to 16). Considering the randomness of sampling, the obtained container scheduling decisions $\mathbf{X}_t$ can violate constraints (12) – (14). If no violations occur, $\mathbf{X}_t$ is valid and we continue the container scheduling process for the next timeslot $t+1$. Otherwise, we require re-sampling $\mathcal{N}_{mig}$ and obtain another $\mathbf{X}_t$, until K trials are finished (line 10) or the scheduling constraints are satisfied (lines 17 to 19).

Finally, we iteratively calculate the container scheduling cost over the time window W for each sample. We choose the container scheduling decisions $\mathbf{X}_t^{min}$ with the minimized scheduling cost among Z samples. According to MPC, we only perform the container scheduling decisions $\mathbf{X}_{t_1}^{min}$ for the first timeslot (lines 20 to 22).

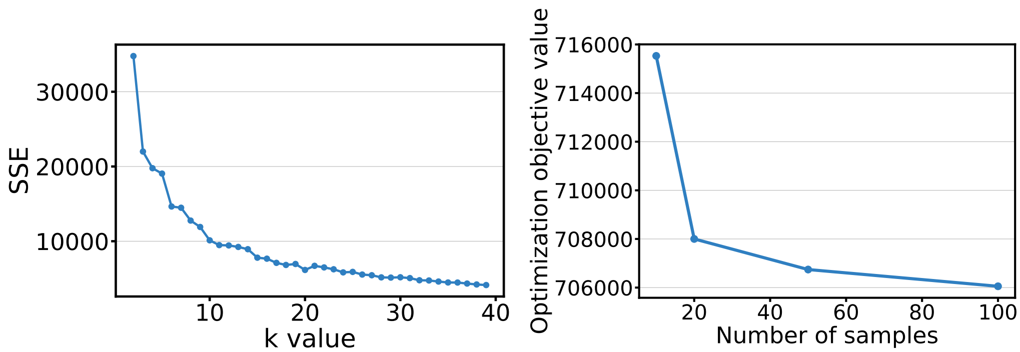
Determining parameters in *Tetris*. We obtain three parameters (*i.e.,* v, W, Z) in Alg. 1 using the methods below. *First* is to determine the sampling rate v for migration containers. We empirically select the proportion of the container resource consumption exceeding $\overline{CPU(t)}$ and $\overline{MEM(t)}$ in migration source servers as the value of v. *Second* is to decide the time window size W. We obtain the maximum time window W_1 where the prediction error of container resource consumption is within the required range (*e.g.,* 15%). Also, we obtain the window size W_2 with the fastest growing number of invalid migrations (in Fig. 3). We finally set W as the minimal value of W_1 and W_2. *Third* is to determine the number of samples Z. To uniformly sample different types of containers, we perform k-means clustering on the CPU and memory resource consumption of containers. We simply use the k clusters as the minimal value of Z.

Time complexity analysis. According to Alg. 1, the time complexity of *Tetris* is in the order of $\mathcal{O}(ZWK \cdot |\mathcal{N}||\mathcal{M}|)$. As the number of containers $|\mathcal{N}|$ and the number of servers $|\mathcal{M}|$ is *far larger* than the time window size W and the number of trials K, the complexity can be reduced to $\mathcal{O}(Z \cdot |\mathcal{N}| \cdot |\mathcal{M}|)$. The computation overhead of *Tetris* is practically acceptable, which will be validated in Sec. 5.4.

5. Performance Evaluation

In this section, we evaluate *Tetris* by conducting prototype experiments based on a 60-container K8s cluster in Amazon EC2 and complementary large-scale simulations driven by a real-world production cluster trace from Alibaba v2018. Our evaluation focuses on answering the following questions:

- *Accuracy:* Can *Tetris* accurately predict the resource consumption of long-running containers? (Sec. 5.2)
- *Effectiveness:* Can our *Tetris* strategy jointly optimize the cluster load balancing degree and migration cost for long-running containers without incurring SLO violations? (Sec. 5.3)
- *Overhead:* How much runtime overhead does *Tetris* bring? (Sec. 5.4)



(a) (b)

Figure 7: (a) Sum of the squared errors (SSE) of k-means clustering on CPU and memory resource consumption of containers with different k values, and (b) the optimization objective value converging at 20 samples in the Monte Carlo method.

5.1. Experimental Setup

Configurations of K8s cluster and *Tetris* parameters. We carry out prototype experiments upon 10 m6a.large EC2 instances. Each instance is equipped with 2 vCPUs and 8 GB memory and initially hosting 6 containers. To obtain complementary insights, we also build a trace-driven simulator based on a discrete-event simulation framework [26] running on a commodity server equipped with 24 3.5 GHz Intel Core i9-10920X CPU cores and 32 GB memory. In addition, we set several key parameters in *Tetris* dynamic model and scheduling strategy using the method in Sec. 4.3. The number of samples Z is set as 20, as the container resource consumption can be clustered with 20 types and the optimization value converges at 20 samples shown in Fig. 7. The time window size W is set as 2 based on the prediction accuracy of container resource consumption and the occurrence of invalid migrations in Fig. 3. The number of trails K and the sampling ratio v of migration containers are empirically set as 10 and 40%, respectively.

Workloads and datasets. We employ three representative workloads in the containerized clusters, including Apache Tomcat server[4], Redis[5], and ResNet50 [27] training. We use Apache-benchmark[6] and redis-benchmark[7] as the time-varying user requests for Tomcat Web server and Redis, respectively. We use CIFAR-10[8] as the dataset for ResNet50. We *randomly* deploy the three workloads on the 60 containers in the K8s cluster. For the prediction of container resource consumption and large-scale simulations, we adopt Alibaba cluster trace v2018 [8] which contains around $4,000$ machines over 8 days. We obtain the CPU and memory resource consumption of $67,437$ containers using linear interpolation and zero-value padding. We set each timeslot as 1 hour and calculate the average resource consumption of containers per hour.

Baselines and metrics. We evaluate *Tetris* against the conventional Sandpiper [7] and the state-of-the-art Metis$^+$ (*i.e.,* a modified version of Metis [3]) scheduling algorithms. To achieve load balancing, Sandpiper performs a greedy worst-fit algorithm to schedule containers according to the container index *volume* calculated by the multi-dimensional resource consumption. Metis$^+$ uses the method in *Tetris* to select migration source servers and migrated containers and leverages the negative value of Eq. (8) as the reward of RL to make container scheduling (*i.e.,* where to place) decisions. In particular, we use the *mean absolute percentage error (MAPE)* [28] to

[4] https://tomcat.apache.org/
[5] https://redis.io/
[6] https://httpd.apache.org/docs/2.4/programs/ab.html
[7] https://redis.io/topics/benchmarks
[8] https://www.cs.toronto.edu/~kriz/cifar.html

Table 2: Number of containers with periodic or aperiodic CPU/memory resource consumption.

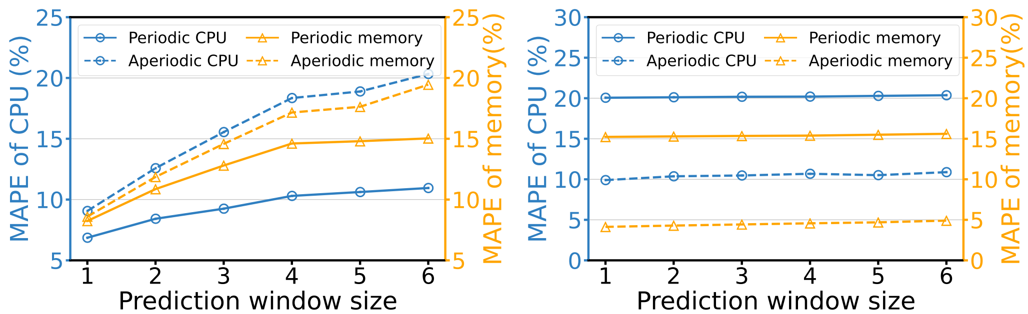
Resource	CPU	Memory
Number of periodic containers	$49,962$	$40,501$
Number of aperiodic containers	$17,475$	$26,936$

evaluate the efficacy of the *predictor* module of *Tetris*. Meanwhile, we adopt the *load imbalance degree* defined in Eq. (1) and the *migration cost* defined in Eq. (5) as well as the number of *SLO violations* to evaluate the efficacy of the *scheduler* module of *Tetris*. Due to the randomness of sampling in Monte Carlo method and workload placement on containers, we illustrate the container scheduling performance with error bars of standard deviation by repeating experiments for three times.

5.2. Predicting Resource Consumption of Containers in Tetris

We first classify the resource consumption of containers as periodic or aperiodic using the method elaborated in Sec. 4.2. As shown in Table 2, the periodic CPU and memory resource consumption accounts for 74.1% and 60.1% of containers, respectively, while the proportion of aperiodic CPU and memory resource consumption are only 25.9% and 39.9%, respectively. Accordingly, most of the containers show a periodic resource consumption over time, which is consistent with a latest measurement study [17].

We next leverage ARIMA and LSTM to predict the periodic and aperiodic container resource consumption. To train the prediction model, we use 70% of the trace data as the training dataset. As shown in Fig. 8, we *first* observe that ARIMA is more accurate than LSTM for predicting the resource consumption of periodic containers. This is because the moving average and



(a) ARIMA (b) LSTM

Figure 8: Average prediction accuracy of CPU and memory resource consumption of $67,437$ containers in Alibaba cluster trace v2018, by varying the prediction window size from 1 to 6.

autoregression in ARIMA can accurately capture the periodic resource consumption of containers, with a moderate prediction error (*i.e.,* less than 15%). *Second,* LSTM achieves a more accurate prediction error (*i.e.,* 5% – 10%) in predicting the resource consumption of aperiodic containers compared with ARIMA as expected. This is mainly because ARIMA can easily be affected by abnormal spikes in aperiodic resource consumption, while LSTM can learn such large resource fluctuations through model training. Interestingly, LSTM fails to predict the resource consumption of periodic containers depicted in Fig. 8(b) simply because LSTM contains nonlinear activation functions and thus overfits the periodic data, thereby causing around 15% – 20% of prediction error for periodic containers [29]. *Third,* the prediction error of container resource consumption with ARIMA can significantly increase from 7.8% to 20.1% with the time window size ranging from 1 to 6. That is the reason why *Tetris* controls the time window size to maintain an acceptable prediction error (*i.e.,* within 15%) of container resource consumption.

In addition, the average prediction time of ARIMA and LSTM are 0.25

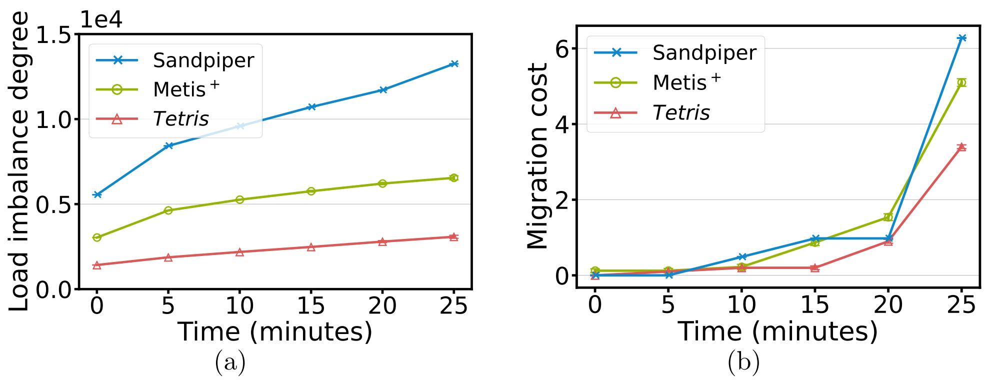


Figure 9: Performance comparison of *Tetris* with Sandpiper and Metis$^+$ strategies under K8s-based prototype experiments, in terms of the cumulative values of (a) cluster load imbalance degree and (b) migration cost of containers over time (*i.e.*, 5 timeslots with 5 minutes each).

seconds and 0.36 seconds, respectively, for each container. As our prediction method in Sec. 4.2 requires a shared model and a personalized model for each container, the average storage footprint of a ARIMA model and an LSTM model are just 556 KB and 908 KB, respectively. Accordingly, such time and space overhead for the prediction of container resource consumption are both practically acceptable.

5.3. Effectiveness of Tetris

Prototype experiments on Amazon EC2. We first evaluate the effectiveness of *Tetris* in a 60-container K8s cluster by setting the *timeslot* as 5 minutes. As shown in Fig. 9, *Tetris* achieves a lower load imbalance degree by 74.4% – 77.8% and 53.0% – 59.6% compared with Sandpiper and Metis$^+$, respectively. Moreover, the migration cost with *Tetris* is 7.8% – 79.5% and 10.3% – 77.0% lower than that with Sandpiper and Metis$^+$, respectively. This is because (1) *Tetris* jointly optimizes the load imbalance degree and migration cost, while Sandpiper greedily migrates containers to alleviate the

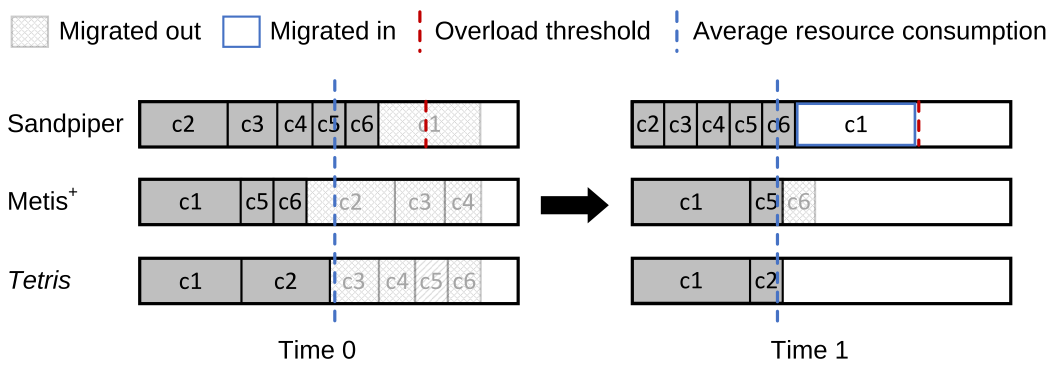


Figure 10: Comparison of container scheduling decisions on server $s1$ of the 60-container K8s cluster in the timeslot $[0, 1]$ under *Tetris*, Sandpiper, and Metis$^+$ strategies.

server hotspot without considering the negative impact of migration cost, which causes 37 invalid migrations in total. (2) Though Metis$^+$ leverages the RL method to explicitly consider the migration cost, it sacrifices the container performance for a certain number of timeslots to optimize container scheduling over the entire time period (*i.e., infinite future*), bringing 12 invalid migrations for 5 timeslots. In contrast, *Tetris* continuously optimizes each timeslot (*i.e.,* 5 minutes) using a certain sliding time window.

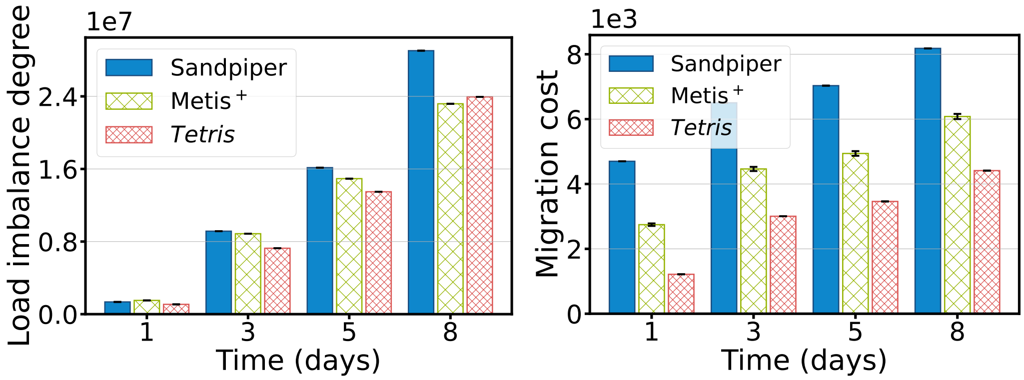
To illustrate the effectiveness of *Tetris*, we further take a close look at the scheduling decisions made by the three strategies. As shown in Fig. 10, we observe that *Tetris* achieves the most load balancing than the other two strategies at both time 0 and time 1, by judiciously migrating out four small containers (*i.e.,* $c3 - c6$) at time 0. In comparison, Sandpiper causes an invalid migration of container $c1$. As server $s1$ is overloaded at time 0 by setting the overload threshold as 85%, Sandpiper chooses to migrate container $c1$ (*i.e.,* the container with the maximum *volume*) to server $s2$. It then migrates container $c1$ back to server $s1$ as server $s2$ is overloaded at time 1, resulting in poor load balancing and large migration cost. As for Metis$^+$, it

Table 3: Cumulative number of SLO violations over time in our prototype experiments on a 60-container K8s cluster.

Duration (timeslots)	0	1	2	3	4	5
Sandpiper	0	0	14	22	26	61
Metis$^+$	0	0	0	8	8	20
Tetris	0	0	0	0	0	0

migrates out containers $c2 - c4$ and $c6$ at time 0 and time 1, respectively, because it adopts online training of the RL model, which has not been trained well enough over the time $[0, 1]$. In addition, Metis$^+$ mainly considers the optimization over the infinite future, which is likely to increase the load imbalance degree or migration cost over a certain time period (*e.g.,* time $[0, 1]$).

We proceed to examine whether *Tetris* can guarantee the SLO of workloads. For simplicity, we consider the containers hosted on the *overloaded* servers as SLO violations. As shown in Table 3, *Tetris* causes zero SLO violations while Sandpiper and Metis$^+$ can cause up to 35 and 12 SLO violations within one timeslot (*i.e.,* at time 5). The reason is that *Tetris* explicitly considers the hard constraints (*i.e.,* Constraints (13)–(14)) on the CPU and memory resource capacities of servers, so as to guarantee the SLO of workloads for each timeslot. However, Sandpiper greedily migrates the container with the largest *volume* to the server with the lightest load. It does not check whether the server load exceeds the capacity on the migration destination server, causing unexpected SLO violations to containers. The RL method in Metis$^+$ is not designed to fully guarantee Constraints (13)–(14). It allows the occurrence of SLO violations over a certain time period to optmize container



(a) (b)

Figure 11: Performance comparison of *Tetris* with Sandpiper and Metis$^+$ strategies on Alibaba cluster trace, in terms of the cumulative values of (a) cluster load imbalance degree and (b) migration cost of containers over 8 days (the timeslot is 1 hour).

scheduling (*i.e.,* maximize the action reward) over the infinite future.

Large-scale simulations driven by Alibaba cluster trace. We next evaluate the effectiveness of *Tetris* with real-world trace-driven simulations, by setting the *timeslot* as 1 hour. As shown in Fig. 11, *Tetris* can reduce the load imbalance degree of the cluster by 9.7% – 28.6% compared with Sandpiper and Metis$^+$. It can also achieve the lowest migration cost, which is reduced by 27.4% – 74.1% than the other two strategies. The experimental results are consistent with our prototype experiments. This is because *Tetris* co-optimizes the load balancing and migration cost to alleviate invalid migrations. In contrast, Sandpiper greedily alleviates the heavy server load using the worst-fit algorithm, resulting in a number of invalid migrations and a high growth rate of migration cost as depicted in Fig. 11(b). Though Metis$^+$ can optimize the load imbalance degree and migration cost over the infinite future, it achieves worse load balance degree in the early stage (*i.e.,* the first 5 days) and better load balance degree in the late stage (*i.e.,* day

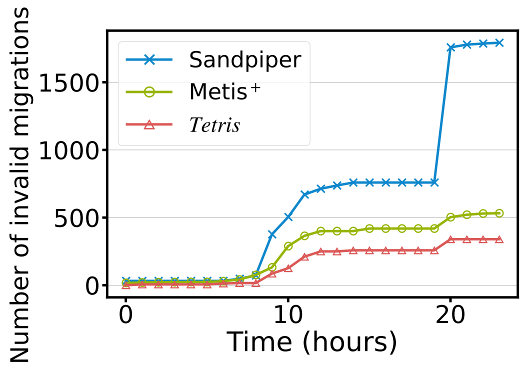


Figure 12: Cumulative number of invalid migrations of *Tetris* compared with that of Sandpiper and Metis$^+$ strategies on a 24-hour Alibaba cluster trace.

8) compared with *Tetris* as shown in Fig. 11(a). This is because it requires continuous online training until the RL model converges (after day 5), which is likely to cause invalid migrations in the early stages.

To verify whether *Tetris* can alleviate invalid migrations, we record the cumulative number of invalid migrations under the three scheduling strategies in our simulation. As shown in Fig. 12, *Tetris* reduces the number of invalid migrations by 59.4% – 81.3% and 32.4% – 78.9% as compared with Sandpiper and Metis$^+$, respectively. *Tetris* and Metis$^+$ can reduce the number of invalid migrations, simply because they both consider long-term co-optimization of load balancing and migration cost for container scheduling. In comparison, Sandpiper only considers short-term optimization of container scheduling, resulting in a significant number of invalid migrations. As Metis$^+$ achieves long-term container scheduling over the infinite future, which can cause more invalid migrations than *Tetris* over a certain number of timeslots. In particular, Sandpiper invokes a significant number of invalid migrations at time 20 because the resource consumption of containers varies greatly and thus a

Table 4: Cumulative number of SLO violations over time in a simulated $67,437$-container cluster driven by Alibaba cluster trace.

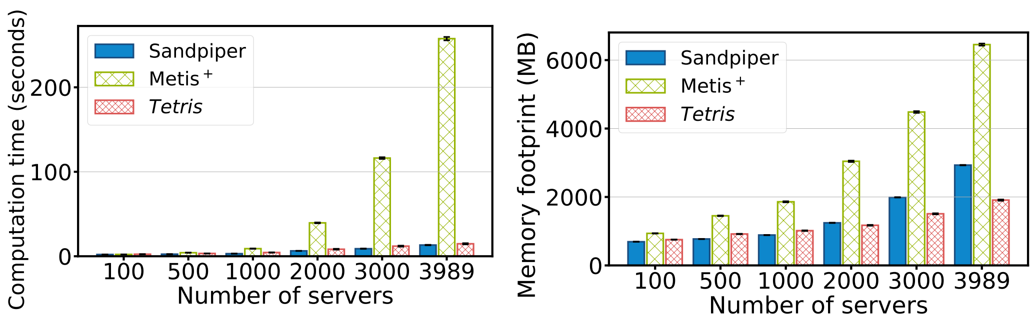
Duration (days)	1	3	5	8
Sandpiper	881	$3,512$	$12,248$	$14,403$
Metis$^+$	0	220	220	918
Tetris	0	0	0	0

number of servers are overloaded at time 20.

Similar to our prototype experiments on Amazon EC2, we examine whether *Tetris* can guarantee the SLO of workloads in large-scale simulations. As shown in Table 4, we observe that *Tetris* does not cause any SLO violations for 8 days, while both Sandpiper and Metis$^+$ can cause up to $14,403$ and 918 SLO violations, respectively, which account for 21.4% and 1.4% of the total number of containers. Our simulation results are consistent with the prototype experiments.

5.4. Runtime Overhead of Tetris

As shown in Fig. 13(a), we observe that the runtime overhead of the three strategies increases *quadratically* with the cluster scale, while Sandpiper and *Tetris* achieves much lower overhead than Metis$^+$. The rationale is that the search space of Metis$^+$ contains all possible mappings of containers to servers, while Sandpiper and *Tetris* only consider a subset of servers and containers, which are the overloaded servers (containers) and the containers on the migration source and destination sets, respectively. Though the time complexity of *Tetris* is in the order of $\mathcal{O}(|\mathcal{N}| \cdot |\mathcal{M}|)$ (as discussed in Sec. 4.3), the *quadratic term ratio* (obtained using polynomial regression) of *Tetris* is only $1.9e-07$, which is much smaller than that of Metis$^+$ (*i.e.,* $2.4e-05$)



(a) Computation time (b) Memory footprint

Figure 13: Runtime overhead of *Tetris* compared with that of Sandpiper and Metis$^+$ strategies for each timeslot (*i.e.,* 1 hour) on a 24-hour Alibaba cluster trace, by varying the number of servers from 100 to 3,989 with each hosting 17 containers.

Table 5: Cumulative computation overhead (in minutes) of *Tetris*, Sandpiper and Metis$^+$ scheduling algorithms executed on a 8-day Aliababa cluster trace.

Duration (days)	1	3	5	8
Sandpiper	4.5	17.6	28.5	46.7
Metis$^+$	86.1	333.6	503.2	827.1
Tetris	6.5	19.8	29.2	51.5

and Sandpiper (*i.e.,* $5.5e-07$). Similarly, *Tetris* consumes much smaller amount of memory than Metis$^+$ and Sandpiper as shown in Fig. 13(b). This is because Metis$^+$ stores the policy of RL which involves all states and actions as well as the large parameters of neural networks. Sandpiper requires storing the *volumes* of all containers and servers. In comparison, *Tetris* only needs to calculate the load imbalance degree of all servers and the migration cost of a subset of containers (*i.e.,* containers to be migrated).

In addition, we illustrate the computation overhead of three strategies over various time scales (*i.e.,* from 1 to 8 days) in Table 5. The computation time of *Tetris* is slightly longer than that of Sandpiper (*i.e.,* 0.7 – 4.8 minutes), while much shorter than that of Metis$^+$ (*i.e.,* 81.6 – 780.4 minutes) as the time scale increases. This is because Sandpiper triggers the fewest algorithm executions only when the resource hotspot occurs. Metis$^+$ greedily selects the largest-reward action over all state spaces in each timeslot, and it also requires online training to update the RL model. In contrast, *Tetris* periodically triggers the execution of Alg. 1 in each timeslot and such time overhead is much smaller than Metis$^+$ per timeslot as depicted in Fig. 13(a). As a result, the runtime overhead of *Tetris* is practically acceptable.

6. Related Work

Short-term optimization of workload scheduling. There have been a number of works devoted to scheduling VMs or containers by considering the *short-term* (*e.g.,* the current or upcoming timeslot) benefits. For example, in order to efficiently consolidate VMs on servers to save energy [30], Dabbagh et al. [31] minimize the energy consumption of migrations by leveraging Wiener filter method to predict resource consumption. UAHS [32] leverages Bayesian optimization to maximize the CPU utilization of servers. To achieve load balancing of clusters, Sandpiper [7] performs the worst fitting algorithm using the server load index based on CPU, memory, and network resources. Lv et al. [20] propose a communication-aware worst-fit decreasing container placement algorithm to optimize the initial container placement. To particularly optimizing the scheduling of long-running workloads, Medea [2] explicitly considers several constraints like affinity and cardinality of containers, while TOPOSCH [33] leverages a prediction-based vertical auto-scaling technique to avoid Quality of Service (QoS) violations and balance the performance of batch jobs. The *short-term* optimization of workload scheduling above can cause invalid migrations as illustrated in Sec. 2.2. In contrast, *Tetris* leverages the MPC approach to achieve the *long-term* optimization of container scheduling and jointly optimize load balancing and migration cost for long-running workloads.

RL-based workload scheduling. Two recent works (*i.e.,* Metis [3], George [1]) design RL algorithms to obtain efficient container placement plans for long-running applications, by modeling the reward as an indicator of container performance and constraint violations. Mondal et al. [4] schedule time-varying workloads to servers based on deep reinforcement learning (DRL) to improve the cluster utilization. To explicitly consider the *long-term* optimization of workload scheduling, Megh [34] leverages an online RL algorithm to perform VM migrations at each timeslot to minimize the energy consumption and avoid service level agreement (SLA) violations. A-SARSA [11] dynamically scales the number of instances based on RL methods to reduce SLA violations. To avoid repeated scheduling due to a fixed scaling upper limit, A-SARSA adjusts the number of actions corresponding to each state. As evidenced by Sec. 2.3 and Sec. 5.3, RL-based scheduling methods can cause unexpected SLO violations by considering the long-term optimization over the infinite future. Also, as shown in Sec. 5.4, RL methods have high time complexity and memory consumption even after the dimensionality reduction, which requires efforts to be deployed in large-scale clusters. To reduce the computation complexity to minutes, *Tetris* leverages the MPC approach to obtain a sub-optimal solution over a certain and sliding time window, which is sufficient for our requirements of *long-term* container scheduling in production clusters.

7. Conclusion and Future Work

This paper presents the design and implementation of *Tetris*, an MPC-based container scheduling strategy to optimize the migration of long-running workloads for achieving load balancing of clusters. By devising a discrete-time dynamic model of shared containerized clusters, *Tetris* judiciously identifies the container migration decisions (*i.e., which and where* to migrate) for each timeslot using the Monte Carlo method, with the aim of jointly optimizing cluster load balancing and migration cost of containers. We conduct prototype experiments on Amazon EC2 and large-scale simulations driven by Alibaba cluster trace v2018. Our experiment results demonstrate that *Tetris* can improve the cluster load balancing degree by up to 77.8% while reducing the migration cost by up to 79.5% with acceptable runtime overhead, compared with the state-of-the-art container scheduling strategies.

As our future work, we plan to extend our discrete-time dynamic model by incorporating more types of server resources such as network and disk I/O resources. We also plan to deploy *Tetris* in large-scale clusters and examine the effectiveness and scalability of *Tetris* container scheduling strategy.

References

[1] S. Li, L. Wang, W. Wang, Y. Yu, B. Li, George: Learning to place long-lived containers in large clusters with operation constraints, in: Proc. of ACM SOCC, 2021, pp. 258–272.

[2] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, S. Rao, Medea: scheduling of long running applications in shared production clusters, in: Proc. of Eurosys, 2018, pp. 1–13.

[3] L. Wang, Q. Weng, W. Wang, C. Chen, B. Li, Metis: learning to schedule long-running applications in shared container clusters at scale, in: Proc. of IEEE SC, 2020, pp. 1–17.

[4] S. S. Mondal, N. Sheoran, S. Mitra, Scheduling of time-varying workloads using reinforcement learning, in: Proc. of AAAI, Vol. 35, 2021, pp. 9000–9008.

[5] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at google with borg, in: Proc. of Eurosys, 2015, pp. 1–17.

[6] D. Bernstein, Containers and cloud: From lxc to docker to kubernetes, IEEE Cloud Computing 1 (3) (2014) 81–84.

[7] T. Wood, P. Shenoy, A. Venkataramani, M. Yousif, Sandpiper: Black-box and gray-box resource management for virtual machines, Computer Networks 53 (17) (2009) 2923–2938.

[8] Alibaba, Alibaba Cluster Trace Program (May 2022).
URL https://github.com/alibaba/clusterdata/

[9] A. Beloglazov, J. Abawajy, R. Buyya, Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing, Future Generation Computer Systems 28 (5) (2012) 755–768.

[10] M. Xu, W. Tian, R. Buyya, A survey on load balancing algorithms for virtual machines placement in cloud computing, Concurrency and Computation: Practice and Experience 29 (12) (2017) e4123.

[11] S. Zhang, T. Wu, M. Pan, C. Zhang, Y. Yu, A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning, in: Proc. of IEEE ICWS, 2020, pp. 489–497.

[12] M. Gaggero, L. Caviglione, Model predictive control for the placement of virtual machines in cloud computing applications, in: Proc. of ACC, 2016, pp. 1987–1992.

[13] M. Gaggero, L. Caviglione, Model predictive control for energy-efficient, quality-aware, and secure virtual machine placement, IEEE Transactions on Automation Science and Engineering 16 (1) (2018) 420–432.

[14] C. E. Garcia, D. M. Prett, M. Morari, Model predictive control: Theory and practice - a survey, Automatica 25 (3) (1989) 335–348.

[15] N. Metropolis, S. Ulam, The monte carlo method, Journal of the American Statistical Association 44 (247) (1949) 335–341.

[16] H. Wu, W. Zhang, Y. Xu, H. Xiang, T. Huang, H. Ding, Z. Zhang, Aladdin: Optimized maximum flow management for shared production clusters, in: Proc. of IEEE IPDPS, 2019, pp. 696–707.

[17] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, Y. Bao, Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces, in: Proc. of IEEE/ACM IWQoS, 2019, pp. 1–10.

[18] M. Xu, C. Song, H. Wu, S. S. Gill, K. Ye, C. Xu, esdnn: Deep neural network based multivariate workload prediction in cloud computing environments, ACM Transactions on Internet Technology (2022).

[19] Y. Zhang, L. Jiao, J. Yan, X. Lin, Dynamic service placement for virtual reality group gaming on mobile edge cloudlets, IEEE Journal on Selected Areas in Communications 37 (8) (2019) 1881–1897.

[20] L. Lv, Y. Zhang, Y. Li, K. Xu, D. Wang, W. Wang, M. Li, X. Cao, Q. Liang, Communication-aware container placement and reassignment in large-scale internet data centers, IEEE Journal on Selected Areas in Communications 37 (3) (2019) 540–555.

[21] E. S. Hou, N. Ansari, H. Ren, A genetic algorithm for multiprocessor scheduling, IEEE Transactions on Parallel and Distributed Systems 5 (2) (1994) 113–120.

[22] M. R. Garey, D. S. Johnson, Computers and intractability, Vol. 174, 1979.

[23] G. E. Box, G. M. Jenkins, G. C. Reinsel, G. M. Ljung, Time series analysis: forecasting and control, 2015.

[24] Z. Li, D. Hoiem, Learning without forgetting, IEEE Transactions on Pattern Analysis and Machine Intelligence 40 (12) (2017) 2935–2947.

[25] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (8) (1997) 1735–1780.

[26] F. Li, B. Hu, Deepjs: Job scheduling based on deep reinforcement learning in cloud data center, in: Proc. of ICBDC, 2019, pp. 48–53.

[27] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proc. of CVPR, 2016, pp. 770–778.

[28] B. L. Bowerman, R. T. O'Connell, A. B. Koehler, Forecasting, time series, and regression: an applied approach, Vol. 4, 2005.

[29] D. Fan, H. Sun, J. Yao, K. Zhang, X. Yan, Z. Sun, Well production forecasting based on arima-lstm model considering manual operations, Energy 220 (2021) 1–13.

[30] Z. Á. Mann, Allocation of virtual machines in cloud data centers – a survey of problem models and optimization algorithms, ACM Computing Surveys 48 (1) (2015) 1–34.

[31] M. Dabbagh, B. Hamdaoui, M. Guizani, A. Rayes, An energy-efficient vm prediction and migration framework for overcommitted clouds, IEEE Transactions on Cloud Computing 6 (4) (2016) 955–966.

[32] C. Luo, B. Qiao, X. Chen, P. Zhao, R. Yao, H. Zhang, W. Wu, A. Zhou, Q. Lin, Intelligent virtual machine provisioning in cloud computing, in: Proc. of IJCAI, 2021, pp. 1495–1502.

[33] R. Yang, J. Zhu, X. Sun, T. Wo, C. Hu, H. Peng, J. Xiao, A. Y. Zomaya, J. Xu, Qos-aware co-scheduling for distributed long-running applications on shared clusters, IEEE Transactions on Parallel and Distributed Systems (2022).

[34] D. Basu, X. Wang, Y. Hong, H. Chen, S. Bressan, Learn-as-you-go with megh: Efficient live migration of virtual machines, IEEE Transactions on Parallel and Distributed Systems 30 (8) (2019) 1786–1801.

Appendix .1. Mathematical Derivations of Eq. (9) and Eq. (10)

To obtain Eq. (9), the load imbalance degree of CPU resource part in Eq. (1) can first be expanded as

$$\begin{aligned}\sum_{i\in\mathcal{M}}\Big(\sum_{k\in\mathcal{N}} x_i^k(t)\cdot cpu^k(t) - \overline{CPU(t)}\Big)^2 = &\sum_{i\in\mathcal{M}}\Big(\sum_{k\in\mathcal{N}} x_i^k(t)\cdot cpu^k(t)\Big)^2 \\ &- 2\cdot\overline{CPU(t)}\cdot\sum_{k\in\mathcal{N}} cpu^k(t) \\ &+ |\mathcal{M}|\cdot\overline{CPU(t)}^2,\end{aligned} \tag{.1}$$

where the last two terms of the expanded expression in Eq. (.1) are actually constant for each timeslot t, which can be denoted by $-|\mathcal{M}|\cdot\overline{CPU(t)}^2$. The load imbalance degree of CPU resource part can be mainly determined by the first term of Eq. (.1), which is given by

$$\begin{aligned}\sum_{i\in\mathcal{M}}\Big(\sum_{k\in\mathcal{N}} x_i^k(t)\cdot cpu^k(t)\Big)^2 = &\,2\cdot\sum_{i\in\mathcal{M}}\sum_{k,l\in\mathcal{N}} x_i^k(t)cpu^k(t)\cdot x_i^l(t)cpu^l(t) \\ &-\sum_{k\in\mathcal{N}} cpu^k(t)^2,\end{aligned} \tag{.2}$$

where the second term is also a constant at time t. Accordingly, the load imbalance degree of CPU resource part can be determined by the first term of Eq. (.2). The load imbalance degree of memory resource part in Eq. (1) is similar to that of CPU resource part by substituting $mem^k(t)$ for $cpu^k(t)$. We formulate $C_b(t)$ as

$$\begin{aligned}C_b(t) = &\frac{2}{|\mathcal{M}|-1}\sum_{i\in\mathcal{M}}\sum_{k,l\in\mathcal{N}} x_i^k(t)x_i^l(t)\cdot\big(cpu^k(t)cpu^l(t) \\ &+\beta\cdot mem^k(t)mem^l(t)\big) + C_1,\end{aligned} \tag{.3}$$

where $C_1 = -\frac{|\mathcal{M}|\cdot\left(\overline{CPU(t)}^2+\beta\cdot\overline{MEM(t)}^2\right)+\sum_{k\in\mathcal{N}}\left(cpu^k(t)^2+\beta\cdot mem^k(t)^2\right)}{|\mathcal{M}|-1}$ is a constant value. Accordingly, Eq. (.3) is actually the cluster load imbalance degree formulated in Eq. (9).

To obtain Eq. (10), we first incorporate Eq. (6) into Eq. (5), $C_m(t)$ can be represented by

$$C_m(t) = \sum_{k\in\mathcal{N}}\sum_{i\in\mathcal{M}} C_{mig}^k(t)\cdot x_i^k(t) - \sum_{k\in\mathcal{N}}\sum_{i\in\mathcal{M}} C_{mig}^k(t)\cdot x_i^k(t)x_i^k(t-1), \tag{.4}$$

where the first term is transformed into a constant $C_2 = \delta\cdot|\mathcal{N}| + \gamma\cdot\sum_{k\in\mathcal{N}} mem^k(t)$, which is related to the memory sum of all containers $mem^k(t), \forall k \in \mathcal{N}$. Accordingly, by incorporating Eq. (7) into Eq. (.4), $C_m(t)$ is determined by the second term of Eq. (.4), which is given by

$$C_m(t) = C_2 - \sum_{k\in\mathcal{N}}\sum_{i\in\mathcal{M}} \left(\delta + \gamma\cdot mem^k(t)\right)\cdot x_i^k(t)x_i^k(t-1). \tag{.5}$$

As a result, Eq. (.5) is actually the migration cost across the cluster formulated in Eq. (10).

Appendix .2. Proof of Theorem 1

Proof. The load imbalance degree $load_i(t)$ of a server $i \in \mathcal{M}$ and Eq. (16) – (17) in Theorem 1 contain both CPU and memory resource consumption of containers. To simplify the proof, we first consider CPU resource of containers (*i.e.*, $cpu^k(t), \forall k \in \mathcal{N}$). By assuming the cluster servers are in the "ideal" load-balanced state with the CPU resource consumption of $\overline{CPU(t)}$ at time t, we can calculate the maximum and minimum values of CPU load imbalance degree of a server (*i.e.*, the sum of pairwise multiplication $mul_{n(t)}$

of $cpu^k(t)$ on a server in Eq. (15)) in Lemma 1 below.

Lemma 1. *Given a server in the "ideal" load-balanced state, which hosts a set of containers with the CPU consumption of* $cpu^1(t), cpu^2(t), \cdots, cpu^{n(t)}(t)$ *and* $\sum_{k=1}^{n(t)} cpu^k(t) = \overline{CPU(t)}$*, where* $n(t) \in [1, |\mathcal{N}|]$*, the sum of pairwise multiplication* $mul_{n(t)}$ *of* $cpu^k(t), \forall k \in [1, n(t)]$ *becomes the maximum (i.e.,* $\frac{\overline{CPU(t)}^2}{2}$*), if and only if* $cpu^1(t) = cpu^2(t) = ... = cpu^{n(t)}(t)$ *and* $n(t) = |\mathcal{N}|$*. Similarly,* $mul_{n(t)}$ *becomes the minimum (i.e.,* $\frac{\overline{CPU(t)}^2}{2} - \frac{1}{2} \cdot \sum_{k=1}^{n(t)} cpu^k(t)^2$*), if and only if* $n(t)$ *is the minimum value that satisfies* $\sum_{k=1}^{n(t)} cpu^k(t) = \overline{CPU(t)}$*.*

Proof. The sum of the pairwise multiplication of $cpu^k(t), \forall k \in [1, n(t)]$ is calculated as

$$\begin{aligned} mul_{n(t)} &= \sum_{k=1}^{n(t)-1} \sum_{l=k+1}^{n(t)} \Big(cpu^k(t) \cdot cpu^l(t) \Big) \\ &= \frac{1}{2} \Big[\Big(\sum_{k=1}^{n(t)} cpu^k(t) \Big)^2 - \sum_{k=1}^{n(t)} cpu^k(t)^2 \Big] \\ &= \frac{\overline{CPU(t)}^2}{2} - \frac{\sum\limits_{k=1}^{n(t)} cpu^k(t)^2}{2}. \end{aligned} \tag{.6}$$

The first term of Eq. (.6) is a constant value. By Cauchy-Buniakowsky-Schwarz Inequality, the second term $\frac{1}{2} \cdot \sum_{k=1}^{n(t)} cpu^k(t)^2$ becomes the minimum (*i.e.*, $\frac{\overline{CPU(t)}^2}{2 \cdot n(t)}$) if and only if $cpu^k(t) = \frac{\overline{CPU(t)}}{n(t)}, \forall k \in [1, n(t)]$. Accordingly, we have

$$mul_{n(t)} \leq \Big(\frac{1}{2} - \frac{1}{2 \cdot n(t)} \Big) \cdot \overline{CPU(t)}^2, \tag{.7}$$

where $\frac{1}{2} - \frac{1}{2 \cdot n(t)}$ is a monotonically increasing function with $n(t) \in [1, |\mathcal{N}|]$. As a result, $mul_{n(t)}$ can reach the maximum when $n(t) = |\mathcal{N}|$, which is given by

$$\max\left(mul_{n(t)}\right) = \left(\frac{1}{2} - \frac{1}{2 \cdot |\mathcal{N}|}\right) \cdot \overline{CPU(t)}^2. \tag{.8}$$

To calculate the minimum value of $mul_{n(t)}$, we have to find the maximum value of the second term $\frac{1}{2} \cdot \sum_{k=1}^{n(t)} cpu^k(t)^2$ of Eq. (.6). According to the perfect square tinomial, the square of the sum of a set of non-negative numbers is larger than or equal to the sum of squares of these numbers. Therefore, we find that $\frac{1}{2} \cdot \sum_{k=1}^{n(t)} cpu^k(t)^2$ can reach the maximum when $n(t)$ is the minimum number to satisfy $\sum_{k=1}^{n(t)} cpu^k(t) = \overline{CPU(t)}$. To find the minimum number of $n(t)$, we sort the CPU consumption of containers in a descending order and greedily select containers from the highest CPU consumption to the lowest. In such a situation above, Eq. (.6) can reach its minimum accordingly.

□

Similarly, we can calculate the maximum and minimum values of memory load imbalance degree of a server (*i.e.,* the sum of pairwise multiplication $mul_{n(t)}$ of $mem^k(t)$ on a server in Eq. (15)). To achieve that, we simply substitute $mem^k(t)$ for $cpu^k(t)$ and $\overline{MEM(t)}$ for $\overline{CPU(t)}$ in Lemma 1. As a result, we combine the CPU and memory resources together as in Eq. (15), and the maximum value of the load imbalance degree of a server in the "ideal" load-balanced state turns out to be Eq. (16). The minimum value of the load imbalance degree of an "ideal" server is calculated as Eq. (17), when such a server hosts the minimum number of containers that satisfies $\sum_{k=1}^{n(t)} cpu^k(t) \leq \overline{CPU(t)}$ and $\sum_{k=1}^{n(t)} mem^k(t) \leq \overline{MEM(t)}$ (*i.e.,* by jointly considering CPU and memory resources) in practice.

Finally, to achieve load balancing, we set the threshold of $load_i(t)$ for migration source server (*i.e.*, $load_{src}(t)$) as the maximum value of the load imbalance degree of an "ideal" server in Eq. (16). Meanwhile, we set the threshold of $load_i(t)$ for migration destination server (*i.e.*, $load_{dest}(t)$) as the minimum value of the load imbalance degree of an "ideal" server in Eq. (17).

$\square$