Heterogeneous Task Co-location in Containerized Cloud Computing Environments

Zhiheng Zhong, Jiabo He, Maria A. Rodriguez, Sarah Erfani, Ramamohanarao Kotagiri, and Rajkumar Buyya
Cloud Computing and Distributed Systems (CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia

***Abstract*—Although cloud computing became a mainstream industrial computing paradigm, low resource utilization remains a common problem that most warehouse-scale datacenters suffer from. This leads to a significant waste of hardware resources, infrastructure investment, and energy consumption. As the diversity in application workloads grows into an essential characteristic in modern datacenters, task co-location of different workloads to the same compute cluster has gained immense popularity as a heuristic solution for resource utilization optimization. Although the existing co-location methodologies manage to improve resource efficiency to a certain degree, application QoS is usually sacrificed as a trade-off when dealing with resource interference between different applications. This paper proposes a containerized task co-location (CTCL) scheduler to improve resource utilization and minimize task eviction rate. Our CTCL scheduler (1) applies an elastic task co-location strategy to improve resource utilization; and (2) supports a dynamic task rescheduling mechanism to prevent severe QoS degradation from frequent task evictions. We evaluate our approach in terms of resource efficiency and rescheduling cost through the ContainerCloudSim simulator. Our experiments with the Alibaba 2018 workload traces demonstrate that CTCL could improve overall resource efficiency and reduce rescheduling rate by 38% and 99% respectively.**

***Index Terms*—Cluster Management, Container Orchestration, Resource Co-location, Resource Heterogeneity, Workload Characterization**

I. Introduction

Cloud computing has become a common industrial computing paradigm, providing convenient service delivery to end-users with quality assurance and cost-efficiency. However, low resource utilization appears as a critical concern in resource efficiency and energy consumption as cloud datacenters grow in scale [1]. Earlier studies have observed that mainstream Cloud Service Providers such as Google, Amazon, and Alibaba, are commonly suffering from poor resource utilization of 40% or under [2]–[4]. With the diversity in application workloads growing into a key feature in modern datacenters, task co-location of different workloads on shared compute resources enjoys a significant rise in popularity for resource utilization improvement and energy saving.

Within cloud computing environments, software systems such as YARN [5] and Swarm [6] support the resource management and job scheduling of compute clusters formed using a set of cloud resources that are dynamically provisioned for application deployments. Unlike traditional cluster management methodologies where different types of applications are assigned to strictly separate compute clusters, modern Cloud-based Cluster Management Systems (CMS) manage to co-locate tasks from workloads to the same cluster for better resource efficiency and cost saving. CMS drives the online task scheduling process, including task placement, resource allocation, performance monitoring, and task migration. Each incoming task might differ in multi-dimensional resource demands (e.g., CPU, memory, storage, network bandwidth, etc.). A representative heterogeneous workload in large-scale CMS is usually divided into two classes [4]: (1) long-running, latency-critical, and user-facing applications prioritized with QoS (Quality of Service) and low-latency assurance; (2) batch-processing jobs assigned to the resource slack left by long-running applications through co-location. For instance, Google Borg manages to save 20-30% physical resources through cell sharing [7], where the idle resources initially assigned to long-running services could be reclaimed for task allocation of lower-priority batch jobs.

However, the performance interference between different applications brought by co-location techniques is still a crucial bottleneck of resource efficiency and QoS assurance. The more heterogeneous applications are allocated to the same compute node, the more each application will suffer from resource contention and performance interference from each other [8]. The highly dynamic and uncertain nature of heterogeneous workloads significantly complicates the orchestration process in terms of balancing the trade-off between resource utilization and QoS requirements. Whenever the prioritized long-running applications experience any critical workload spikes, the co-located batch jobs could be potentially differed, evicted, or rescheduled as a victim of performance interference and resource overloading caused by poor co-location decisions. Though task eviction and rescheduling, where tasks are terminated and moved back to the application queue, are designed as a complementary mechanism to alleviate potential performance interference, QoS degradation in lower-priory applications remains an inevitable side effect. To minimize rescheduling rate, CMS should improve the efficiency and accuracy of initial task placement decisions. Detailed workload behaviours of co-located applications should be evaluated during task allocation.

To address these issues, container technologies are widely accepted in task co-location with strong resource and performance isolation. As a lightweight application virtualization

2375-5261/20/$31.00 ©2020 IEEE
DOI 10.1109/ISORC49007.2020.00021

technology, containers provide a logical packing mechanism for application abstraction that packages software and dependencies together. Compared to virtual machines (VM) which support resource virtualization at the hardware level, containers offer a virtual runtime environment based on a single operating system (OS) kernel and emulate an OS instead of booting an entire OS for each application. Hence, containerized compute clusters enjoy higher environmental consistency, resource isolation, portability, and scalability in contrast with VM-based clusters [9]. These features lead to the continuously rising popularity and adoption of this technology. For example, Borg develops its co-location strategies through leveraging resource isolation offered by Linux cgroups-based containers [10], while Alibaba designs a two-level CMS architecture on top of Fuxi and Sigma schedulers under a semi-containerized environment [11].

For further improvement of the task co-location process in an interference-aware manner, workload characterization of containerized applications plays an important role in identifying their resource consumption patterns and thereby avoiding potential interference between applications by making better task placement decisions [12]. Tasks with similar resource usage patterns (e.g., CPU and memory) could be classified into the same class through an off-the-shelf algorithm such as K-means++ [13]. We aim to apply workload characterization techniques under the context of online task scheduling without a priori knowledge of incoming tasks.

To ensure application QoS requirements, CMS should be aware of workload behaviours before making co-location decisions. Hence, it is necessary to build a proper workload characterization solution that could accurately predict the resource usage of higher-priority applications and accordingly manage the co-location of batch jobs. As for task rescheduling, CMS should also dynamically adjust resource configurations according to runtime performance metrics instead of assuming a static resource demand and task duration. To meet these requirements, we propose the containerized task co-location (CTCL) scheduler and make the following two **key contributions**:

1) It applies an elastic task co-location strategy to improve resource utilization and alleviate performance interference;
2) It supports a dynamic task rescheduling mechanism to prevent severe QoS degradation from frequent task evictions.

The rest of the paper is organized as follows: Section II introduces the background and related work of workload characterization and task co-location in modern CMS. Section III formalizes the problem definitions, while the orchestration algorithms in CTCL are described in Section IV. In Section V, we present the experimental configurations and evaluation results of the proposed approach. Finally, conclusions and future work are given in Section VI.

II. Background and Related Work

A. Workload Characterization

Experience studies: Considering the diversity and dynamicity of heterogeneous workloads undertaken by modern datacenters, workload characterization is an essential step for CMS to optimize initial task placement decisions and dynamically adjust configurations at runtime for improving resource utilization [14]. Previous studies manage to characterize workload behaviours at the infrastructure level in different ways, such as building resource usage patterns for VMs/physical machines [15]–[17]. Within a compute cluster deployed with heterogeneous workloads that differ in behaviour patterns, machine-level resource utilization profiling could be limited in accuracy. Continuous monitoring of application-level metrics should be considered for further behaviour identification and QoS management [18]. There are also multiple works focusing on workload modeling at a single-application level using various approaches such as Signal Processing [19], Neural Network, and Linear Regression [20]. However, their solutions are built on the assumption of a priori knowledge of application models and cannot manage the scale of workloads in existing cloud datacenters, which need to accommodate tens of thousands of potentially unknown applications on a daily basis.

Cluster trace analysis: To better understand the nature and characteristics of workloads from real cloud environments, cluster traces from leading Cloud Service Providers have been analyzed in various studies. Google cluster traces have been widely referenced for analysis of application-level workload behaviours in warehouse-scale datacenters, regarding task placement constraint, resource demand, priority, lifetime, and heterogeneity of server configurations, etc [21]–[23]. Furthermore, there have been studies with reference to heterogeneous workload categorization and behaviour identification through many algorithms, among which K-means is popular for analyzing workload characteristics and task classification [12]. However, most of them only apply K-means for workload characterization based on offline analysis. In order to manage the container orchestration process, it is imperative to update workload characterization for containerized applications dynamically and accurately. In this work, we utilize K-means++ for task classification, since it improves both accuracy and speed compared with K-means [13], [24]. Although there are some improvements for K-means++, it owns competitive performance with most of its derivatives [25].

B. Task Co-location in Cloud-based Cluster Management Systems

As summarized in Table I, a series of works have been proposed to manage the co-location process of heterogeneous workloads in modern CMS. A centralized scheduler model is commonly utilized by monolithic CMS to manage task assignment and rescheduling decisions such as Borg [7]. Borg leverages the resource isolation provided by containerized applications to minimize performance interference. By contrast, Mesos builds a two-level CMS architecture that supports resource sharing across different frameworks through negotiation

TABLE I
A COMPARISON OF RELATED WORKS WITH OURS.

CMS Policy	Preemption	Rescheduling	Resource Reclamation	Workload Characterization	Load Prediction	Heterogeneous Autoscaling
Borg [7]	✔	✔	✔	✖	✖	✖
Mesos [26]	✖	✖	✔	✖	✖	✖
Omega [27]	✔	✔	✖	✖	✖	✖
Apollo [28]	✔	✔	✔	✖	✖	✖
Alibaba [4]	✔	✔	✔	✖	✖	✖
Kubernetes [18]	✔	✔	✔	✖	✖	✖
Paragon [29]	✖	✔	✖	✔	✖	✔
CTCL	✔	✔	✔	✔	✔	✔

between framework-specific schedulers and the master process [26]. Omega manages to minimize task scheduling time by maintaining multiple shared-state schedulers, each of which keeps a global view of the underlying compute cluster [27]. Similarly, Apollo supports distributed schedulers in a loosely coordinated manner [28]. Alibaba proposes a hybrid CMS architecture of two levels and shared-state scheduling, where Sigma [4] handles containerized long-running services and Fuxi [30] allocates non-containerized batch jobs directly on the physical machine layer. Besides, a global controller tracks the status of the whole cluster and coordinates co-location decisions.

These state-of-the-art works commonly adopt heuristic methods in task scheduling to improve resource utilization, such as overprovisioning, overbooking, and overcommitment [31], without a robust workload characterization solution. This usually limits the accuracy of co-location decisions, due to lack of consideration of detailed workload behaviours at runtime. Task preemption and rescheduling are thereby utilized as complementary mechanisms to alleviate potential performance interference from poor co-location arrangements. Nonetheless, most of the existing rescheduling algorithms are implemented under the assumption of a static resource demand, which might lead to task misplacement and severe QoS degradation. Frequent reallocation of lower-priority tasks has become a critical threat in QoS management. The quality of the initial task placement during co-location could significantly impact resource utilization, rescheduling costs, and overall system performance.

To address these issues, Delimitrou et al. [29] introduced the Paragon QoS-aware scheduler by applying an offline training algorithm for workload classification based on historical log analysis at large scale. However, their 1-minute profiling mechanism can not accurately identify workloads with significant performance deviations over long periods of time, which may lead to sub-optimal scheduling decisions. Moreover, Paragon does not consider latency-critical or multi-priority applications in its workload scenarios. In other words, it is not capable of providing QoS assurance to higher-priority applications against unexpected performance interference. To avoid inefficient resource allocation caused by inaccurate initial task configuration, Kubernetes [18] supports dynamic task configuration adjustment through vertical autoscaling, task redeployment, and on-demand node provisioning. However, not all types of workloads could tolerate redeployment. In this work, we propose an interference-aware scheduler for efficient task co-location with a dynamic rescheduling algorithm without assuming a fixed resource demand or task duration.

III. PROBLEM FORMALIZATION

This section enumerates our assumptions for container-based applications and resources, followed by the problem definition.

A. Assumptions

CTCL acts as a CMS scheduler of a containerized compute cluster like YARN [5] or SWARM [6], deployed on top of physical machines, where containerized applications could be submitted by multiple users simultaneously with specific configurations of requested resources.

Workload: We consider a heterogeneous workload model mainly consisting of the following two types of containerized applications:

1) Long-running services that undertake latency-critical requests and require high availability, QoS assurance, and low latency, such as search engines, web servers, and databases. These jobs usually request/reserve more compute resources than they actually need in case of any unexpected workload spikes.
2) Transient batch jobs that have a limited lifetime with looser performance requirements. These jobs are configured with lower priorities in task scheduling and are more tolerant to short-term performance fluctuations. The actual resource utilization of batch jobs could potentially exceed its requested volume at runtime [4].

The two most typical application workloads referenced in the existing CMS are originally from Borg and Alibaba. Our motivation is to build a robust task scheduling mechanism in terms of resource utilization optimization and task rescheduling rate reduction.

Resource: Containerized applications are assumed to be deployed directly on physical machines within the compute cluster, which saves the platform virtualization cost compared with VM-based CMS solutions.

1) The datacenter provides heterogeneous physical machines that differ in multiple dimensions: sizes (CPU,

RAM, disk), processor type, performance, and capabilities. These compute nodes share the same geographic location and datacenter network, while the internal communication cost between nodes is low.

2) An application could have different execution times and performance on different machine types. For instance, a memory-intensive task could run faster in a high-memory machine, while a network-intensive task may have specific requirements regarding bandwidth and network latency.
3) Compute instances could be scaled up/down through autoscaling of extra instances and brownout techniques [32], [33]. It may take a few minutes to boot up a new instance and merge it into the current cluster. Such a time gap is referenced as instance acquisition lag in our work [34], [35]. By contrast, scaling down a running instance only takes a few seconds.

B. Problem Definition

To manage the task co-location process of heterogeneous workloads in an interference-aware manner, it is a basic requirement to rapidly and precisely categorize incoming applications based on their resource consumption (e.g., CPU and memory). In such a way, we can evaluate potential performance interference between co-located applications. Therefore, our primary goal is to implement an online task co-location algorithm for warehouse-scale datacenters without a priori information of submitted tasks.

Workload characterization: Instead of modeling each incoming workload through long-term behaviour monitoring, our approach starts from characterization of historical workload traces in the compute cluster. The new workload model could thereby be expressed as a combination of known application classes. The K-means++ algorithm [13] is adopted for task classification among the historical traces, with reference to workload dimensions such as CPU usage, memory consumption, disk storage, and network bandwidth.

Behaviour identification: For each newly deployed task S, its behaviour pattern B is identified through empirical observation of its resource utilization trace within a configurable period p. We calculate the distances between every center given by K-means++ algorithm and the new utilization trace of the underlying task in p, while the closest center is regarded as its class for workload characterization. This process has time complexity of $O(dk)$. The resource utilization is then accordingly estimated based on B at runtime. The new task traces will be periodically integrated into the dataset for reclustering by K-means++ in a cycle l, with the existing centers as initial ones. The time overhead of reclustering is still $O(ndki)$, where n is the size of the new dataset and i is usually smaller than that from scratch. Within the cycle l, a new trace is classified into the closest center under limited empirical observation time p. The shorter p is, the higher the chance of misestimation will be. Hence, there is a trade-off between the accuracy for classifying the new trace and the length of observation time. As batch jobs are usually short and unpredictable in behaviour [4], [7], the scope of workload characterization and behaviour identification in our work is limited to long-running services.

Scheduling: Each incoming task S_i submitted at time point t is configured with multi-dimensional resource demands D_i. The scheduling algorithm decides the best-fit resource allocation to compute node P_v for S_i:

$$Schedule_t = \{S_i \rightarrow P_v\} \tag{1}$$

During the scheduling process of each new long-running service, the available resource volume A_v on each node P_v is evaluated based on its capacity C_v and the resource allocation R_i of each active task ak_i already deployed on P_v:

$$A_v = C_v - \sum_{ak_i \in S_v} R_i \tag{2}$$

As for each incoming batch job, it is allowed to use the idle resources I_i left by each existing long-running service L_i in addition to A_v in Equation 2. I_i is predicted based on its corresponding workload characterization class B_i within a configurable time interval l:

$$I_i = R_i - \max_{l}(B_i) \tag{3}$$

$$A'_v = A_v + \sum_{L_i}^{L_v} I_i \tag{4}$$

To ensure successful deployment of task S_i, it will only be allocated to a node with at least its required amount of resources D_i available, while the amount of resources in the above equations is evaluated as a vector of multi-dimensional task confirmations such as CPU, memory, and storage.

Scaling: It is decided by the scaling mechanism how many physical machines N_v of each machine type P_v should be scaled up/down at time point t in response to the workload fluctuation within the cluster:

$$Scaling_t = \{P_v, N_v\} \tag{5}$$

During the evaluation process of autoscaling, each instance type with n-dimensional resource configurations is associated with a resource efficiency score defined by its normalized used constraining resource nc [36]. nc is decided by the usage of each resource type r_i and its corresponding weight w_i within the constraining scope:

$$nc = \sum_{i=1}^{n} r_i * w_i \tag{6}$$

Rescheduling: If multiple tasks are competing for the same bottleneck resources simultaneously in compute node P_v at time point t, the rescheduling algorithm will decide a portion of its lower-priority tasks S_v to be evicted and placed to other available instances:

$$Rescheduling_t = \{P_v, S_v\} \tag{7}$$

The rescheduling cost T_i of task S_i is defined by the task progress loss (task duration before eviction) pl and task

reallocation (redeployment to another available node) time ra:

$$T_i = pl + ra \tag{8}$$

If a task is rescheduled multiple times n, the total cost is:

$$T_{S_n} = \sum_{i}^{n} T_i \tag{9}$$

Rescheduling is designed as a complementary method to handle resource overloading and performance interference in response to potential resource mismatch in initial task allocation caused by: (1) misestimation in workload characterization of long-running services; (2) large gap between the requested and actual resource usage of batch jobs. Therefore, the resource configuration R_i of a task S_i should be adjusted to U_{i_j} during rescheduling, according to its maximum resource usage U_i during its task duration t instead of the original resource demand:

$$R'_i = \max_{j \in t}(U_{i_j}) \tag{10}$$

The available resource volume for each rescheduled task is also evaluated through Equation (4), while the time interval in resource usage prediction is set to its maximum task duration $max(t)$. These configuration adjustments are implemented to ensure that the rescheduled task is assigned with enough resources within its duration and thereby prevent multi-rescheduling cases.

Overall, our primary goal is to find the maximum resource utilization rate U_t and minimum cluster size Z_t at time point t with S_t existing tasks and P_t machines through the aforementioned methods:

$$\max(U_t) = \max(\frac{\sum_i^{S_t} U_i}{Z_t}) \tag{11}$$

$$\min(Z_t) = \min(\sum_{i}^{P_t} Z_i) \tag{12}$$

IV. Container Orchestration Algorithms

This section introduces the orchestration algorithms for task placement and cluster size adjustment.

A. Scheduling Algorithm

The efficient initial placement of a task is a key concern in terms of resource utilization, QoS assurance, and overall system performance [14]. Every new task is kept in a pending queue by default. CTCL periodically picks tasks from the queue and allocates them to nodes with enough requested resources through a configurable scheduling cycle sc. For each pending task, a two-phase selection process is designed to determine where it will be deployed:

1) Filtering: The first decision made by the scheduler is to select the nodes with enough available resources defined by Equation (2) or (4), depending on its application type.
2) Ranking: As there is usually more than one node selected during the filtering phase, these nodes are then scored and ranked by a configurable set of priority methods. To minimize the chance of task rescheduling caused by potential resource overloading or performance interference between co-located applications, the Least Requested Priority (LRP) algorithm (shown in Algorithm 1) is utilized as the default priority method during ranking. The node with the lowest resource utilization is most preferred for task allocation in LRP. Compared with other prevalent approximation algorithms for online bin packing such as First Fit Decreasing (FFD) or Best Fit Decreasing (BFD), LRP has the effect of spreading tasks across nodes for load balance and application protection against possible resource shortage.

Algorithm 1 Least Request Priority

Input: Pending task p and node group ng
Output: Scheduling plan S

```
select n, min(n.utilization) from ng
where n.availableResource ≥ p.resourceDemand
if n ≠ NULL then
    S = {p → n}
    return S
else
    return NULL
end if
```

For a pending task T_p, the available resource volume R at node P with T_e existing tasks is evaluated under time complexity of $O(T_e)$.

B. Rescheduling Algorithm

When a node suffers from severe resource overloading and performance interference, the Shortest Runtime Rescheduling (SRR) algorithm (shown in Algorithm 2) is referenced for decision making of a portion of its active batch jobs to be evicted and reallocated to other available nodes. SRR is chosen to minimize the QoS degradation during the rescheduling process. To reduce the task progress loss and overall cost as described in Equation (8), the rescheduling process starts from the active job with the shortest current lifetime (most recently deployed) until the utilization of the underlying node drops under a configurable threshold T, where the overall resource utilization is defined as a vector of multiple resources such as CPU, memory, and network bandwidth. The task reallocation process uses the same LRP algorithm, where the resource demand of the underlying task is updated to its maximum resource usage before eviction, instead of assuming a fixed resource demand as given in its original task specification.

C. Scaling Algorithm

If no scheduling plan can be produced due to insufficient resources, autoscaling will be invoked accordingly. The Greedy Autoscaling (GA) algorithm (shown in Algorithm 3) is implemented to find a combination of heterogeneous machines with the highest efficiency scores in terms of the requested resource volume by iterating through the available instance flavours multiple times. On the other hand, if a node stays

Algorithm 2 Shortest Runtime Rescheduling

Input: Overloaded node o, node group ng, and overloading threshold T

```
batches ← active batch jobs on o sorted by o.runtime in ascending order
av ← ng − o
for (batch in batches) do
    evict(batch)
    LRP(batch,av)
    if (o.resourceUtilization < T) then
        break
    end if
end for
```

idle with no tasks deployed on it for a configurable period (five minutes by default), it will be scaled down to avoid resource wastage.

Algorithm 3 Greedy Autoscaling

Input: Pending tasks pts in descending order of resource demands and available instance flavours f

Output: Instance combination ic

```
while (pts.size > 0) do
    f_i ← flavour with the highest resource efficiency score max(score_f)
    for (p in pts) do
        if (f_i.availableReource >= p.resourceDemand) then
            bind(f_i, p)
            pts.remove(p)
        end if
    end for
    ic.append(f_i)
end while
return ic
```

For scaled instance v, the node provisioning time T_v is the sum of flavour scoring time T_s, task binding time T_b, and instance acquisition lag L:

$$T_v = T_s + T_b + L \tag{13}$$

Assuming n instances are concurrently scaled up in response to p pending tasks, the overall time overhead T_a of autoscaling is:

$$T_a = \max_{i \in n}(T_i) \tag{14}$$

V. Performance Evaluation

We evaluated the proposed approach through ContainerCloudSim [9] simulation toolkit using Alibaba cluster workload by comparing the performance with the original traces and Kubernetes [18] scheduling policies.

A. Workload

We use the Alibaba 2018 cluster traces [37] to replay the heterogeneous workload, which provides detailed configurations and runtime information of over 9K long-running services and 4M batch jobs on 4K physical machines in eight days. All the physical machines in Alibaba traces (AT) are homogeneous with 96 CPU and one unit of memory (normalized for confidential reasons). Unlike Google traces, AT provide a clear specification of task configurations, including the initial resource demand, duration, and node allocation. Besides, AT record the runtime resource consumption throughout the lifecycle of each task, including CPU, memory, network, IO, and MPKI. To monitor the potential resource mismatch and inaccurate scheduling decisions, AT also keep track of tasks that are evicted and rescheduled due to performance interference. Therefore, we can simulate the lifecycle of each task and machine by periodically updating their resource usage.

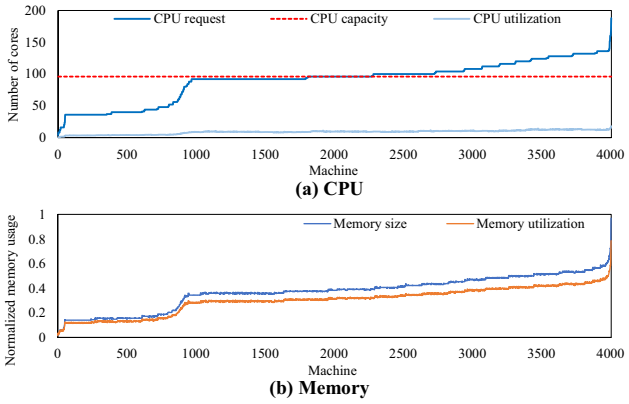

Fig. 1. Resource reservation of long-running services.

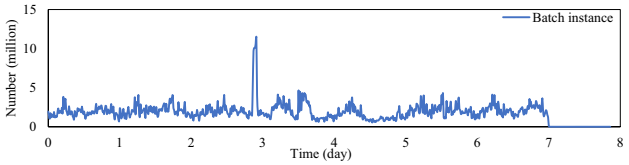

Fig. 2. Batch instance arrival rate.

Fig. 1 depicts the resource reservation of long-running services in each machine (sorted in ascending order) in terms of resource demands and average utilization. It is quite common for long-running services to reserve extra resources for handling unexpected workload spikes with QoS assurance. In most machines, the total CPU request is almost equal to or exceeding the CPU capacity of 96 cores, as Sigma [1] allows a certain level of resource overbooking. Nonetheless, the actual CPU utilization of these tasks is very low (9.5% on average). By contrast, they enjoy higher average memory usage of 80.6%. Overall, long-running services are memory-intensive and CPU-inactive.

Each batch task in AT consists of duplicate instances with the same application logic and resource demand. As shown in Fig. 2, the arrival rate of batch instances follows a nocturnal

pattern where it is higher during nighttime. Note that the workload density on the eighth day is quite low with almost no batches submitted. As addressed in [4], batch jobs in AT are usually short (seconds to minutes) and unpredictable in resource utilization. Most batch jobs actually use more resources at runtime than requested. Therefore, the high throughput and unpredictability of batch jobs could lead to potential resource mismatch in task co-location and frequent task rescheduling. Our experiments demonstrate how these issues could be addressed under different scheduling policies. To scale the workload size to fit with the experiment environment, we randomly choose 25% of the machines in AT and the application workloads deployed on them in a consistent period of eight days. Although AT provide multi-dimensional task configurations, some metrics (e.g., disk storage and network traffic) show low demands with little variation over time. Therefore, we only consider CPU and memory usage within the scope of our experiments.

B. Simulation

Considering the large scale of AT, we choose to evaluate the performance of the proposed approach through ContainerCloudSim [9] simulator with our scheduling policies implemented as an extension. It is a toolkit designed for simulation of containerized cloud computing environments where container orchestration strategies could be evaluated in a repeatable and scalable manner, such as task placement, rescheduling, and resource overbooking. ContainerCloudSim is built on top of CloudSim [38] simulator, which provides a generic simulation framework for modeling cloud computing infrastructures and applications.

We first replay the workload traces following the original application deployment decisions as a baseline in our evaluation. Since the memory capacity of each homogeneous machine in AT is normalized to one unit for confidential reasons, we reference the following instance types belonging to Alibaba Bare Metal Instance Family [39] in our simulation environment. $ecs.ebmg5s.24xlarge$ is assumed to be the default instance type used in AT.

TABLE II
INSTANCE CONFIGURATIONS.

Instance type	CPU	Memory (normalized)	Bandwidth
ecs.ebmc5s.24xlarge	96	0.5	30 Gbit/s
ecs.ebmg5s.24xlarge	96	1	30 Gbit/s
ecs.ebmr5s.24xlarge	96	2	30 Gbit/s

The Kubernetes (K8s) scheduling policies are also included in our evaluation as a benchmark with their algorithms implemented in ContainerCloudSim. The default K8s scheduler utilizes a similar LRP algorithm to manage initial task placement decisions, where all types of jobs are treated equally regardless of their specific characteristics. To prevent inaccurate resource allocation during scheduling, K8s develops a vertical application autoscaling mechanism [40] where applications' configurations are dynamically adjusted according to their resource utilization at runtime. If a task actually utilizes far more resources in its deployed node, it will be rescheduled to other available nodes. For any pending tasks failing to be scheduled due to cluster-level resource shortage, K8s Cluster Autoscaler (CA) could scale up a homogeneous instance per time on an on-demand basis. $ecs.ebmg5s.24xlarge$ is also configured as the default instance type in K8s.

The proposed scheduler CTCL manages to replay the workload traces with heterogeneous compute resources (all the instance types in Table II) with the simulation parameters summarized in Table III. As the workload traces in AT are preprocessed in a 15-minute window during task classification, the prediction time interval is set to 15 minutes. The instance acquisition lag is generated under a normal distribution between 4 to 7 minutes, while the overall resource utilization threshold in rescheduling is set to the same figure of 80% as AT. Since AT follow a daily pattern, the reclustering cycle is defined as 24h. To evaluate how the accuracy of task placement and resource efficiency could be impacted by the length of empirical observation time (EOT) before behaviour identification, we use different observation times $(12, 24h)$ in our experiments. Based on the average cluster-level resource utilization of AT addressed in [4], the weights of CPU and memory are configured as 0.38 and 0.62 respectively during the evaluation process of scheduling and autoscaling.

TABLE III
SIMULATION PARAMETERS.

Parameter	Value
Decision making time of task placement	20ms
Container startup time	300ms
Task eviction time	2ms
Time interval in utilization prediction	15min
Instance acquisition lag	4-7min
Reclustering cycle	24h
Empirical observation time	12, 24h
Resource utilization threshold in rescheduling	80%
Weight of CPU	0.38
weight of memory	0.62

C. Results

This section describes our experimental results in terms of workload characterization and resource efficiency.

Workload characterization: The historical workload traces in AT are recorded within a consistent period of 8 days with various lengths. We preprocess them by dividing each day into 96 uniform intervals ($[0, 15min, 30min, \cdots, 1385min]$), with the closest utilization rates regarded as those at that time. In the end, every trace is summarized using 96 elements with each selected from the highest utilization in 8 days. We run K-means++ with 10 different center seeds and select the best one in order to guarantee the stability of clustering in every cycle l. The hyperparameter k is selected in the range of $[10, 15, 20, 25, 30]$, since workload characterization using K-means++ is not trivially sensitive to k. When $k = 20$ for workload characterization of both CPU and memory utilization

on AT, the clustering results perform the best in the following evaluation of resource efficiency and task rescheduling rate. As shown in Fig. 3, the CPU utilization of most workload classes follows a diurnal pattern where it is obviously higher during daytime (9:00 - 21:00). On the other hand, Fig. 4 depicts that the memory usage is relatively stable without dramatic fluctuation over time, as most long-running services are memory-intensive.

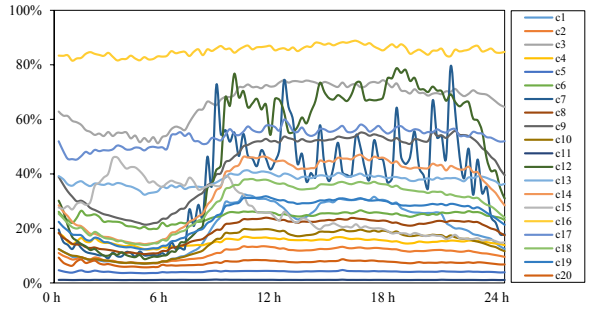

Fig. 3. Clustering results of CPU utilization ($k = 20$).

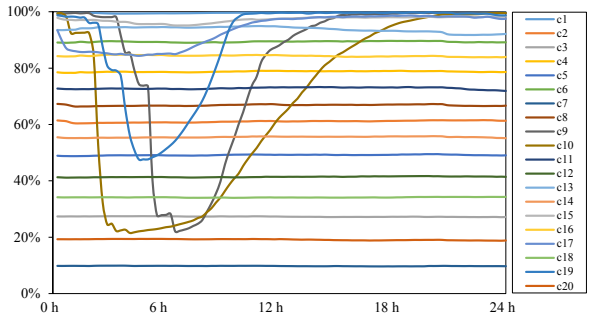

Fig. 4. Clustering results of memory utilization ($k = 20$).

Resource efficiency: The workload traces are repeated for ten iterations under each scheduling policy to calculate the average values of any significant figures for verification of the results with higher validity. Hence, all test results shown are mean values over ten runs. Each scheduling policy is primarily evaluated based on CPU utilization, memory utilization, and cluster size (number of active nodes). Figs. 5-7 depict these metrics under different scheduling policies, including the original AT, K8s, and CTCL. A summary of the overall resource efficiency is given in Table IV. As there is no significant variation in these metrics under different workload characterization classes, class-20 ($k = 20$) with different empirical observation times (12h and 24h) is chosen as the best-performance case included in these figures.

As depicted in Table IV, CTCL outperforms the other scheduling policies in terms of resource efficiency. Compared with AT, CTCL improves the average CPU and memory utilization by at least 62% and 12% respectively through efficient task co-location and heterogeneous autoscaling. Based on the workload characterization results of long-running services,

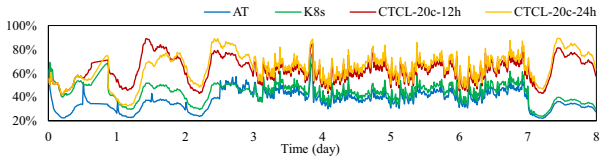

Fig. 5. Overall CPU utilization of the cluster within 8 days.

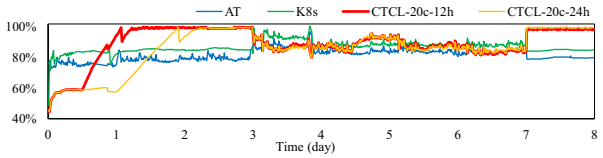

Fig. 6. Overall memory utilization of the cluster within 8 days.

CTCL estimates the resource slack left by them through workload prediction and thereby manages co-location of batch jobs more accurately. Although K8s achieves better overall performance compared with AT through dynamic task configuration adjustment, its Cluster Autoscaler only utilizes homogeneous machines during autoscaling, ignoring the characteristics of pending tasks. This becomes a bottleneck in resource efficiency management, which could lead to potential resource fragmentation. Fig.5 proves that the CPU utilization of K8s (41% average) is only slightly higher than AT (39% average). By contrast, CTCL manages autoscaling more efficiently using heterogeneous machines, according to their resource efficiency scores. As shown in Fig. 7, it can also scale down the idle machines to prevent resource wasting when workload density drops down. Therefore, the average cluster size of CTCL is 40% and 37% smaller than AT and K8s respectively.

Since CTCL needs an empirical observation period to collect resource usage metrics for accurate behaviour identification, the idle resources reserved by long-running service

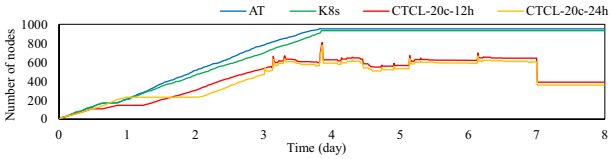

Fig. 7. Changing cluster size within 8 days.

TABLE IV
OVERALL RESOURCE EFFICIENCY.

Algorithm	Average CPU utilization (%)	Average Memory utilization (%)	Maximum cluster size
AT	39	78	952
K8s	41	83	933
CTCL (20c-12h)	63	90	808
CTCL (20c-24h)	65	87	777

could be left unused then. As shown in Fig. 6, the memory utilization in each CTCL experiment (with EOT in (12h,24h)) is actually lower than AT and K8s during the initial EOT. Only after the first round of behaviour identification, task allocation is allowed by CTCL scheduler. Hence, we can observe obvious utilization improvement after the initial EOT. There is also a trade-off between resource efficiency and classification accuracy decided by the length of EOT. Compared with 20c-12h, the utilization of 20c-24h is much lower in the first two days as a result of longer EOT. During the rest six days, 20c-24h slightly outperforms 20c-12h in utilization through more accurate behaviour identification and workload prediction. The similar patterns can also be observed in CPU utilization and cluster size adjustment. Therefore, a proper configuration of EOT should balance these metrics to improve the overall performance.

TABLE V
BATCH INSTANCE RESCHEDULING RATE.

Algorithm	Rescheduled instance	Multi-rescheduled instance	Rescheduling cost (s)
AT	210503	42366	214
K8s	30422	0	175
CTCL (10c-12h)	2735	0	183
CTCL (15c-12h)	2420	0	185
CTCL (20c-12h)	1968	0	168
CTCL (25c-12h)	2003	0	186
CTCL (30c-12h)	1972	0	180
CTCL (10c-24h)	2316	0	179
CTCL (15c-24h)	2221	0	171
CTCL (20c-24h)	1894	0	170
CTCL (25c-24h)	1931	0	180
CTCL (30c-24h)	1900	0	172

The rescheduling rates of all the evaluated solutions are presented in Table V. AT suffer from the highest rescheduling rate and cost, while over 20% of the rescheduled batch instances are reallocated multiple times. As Fuxi manages task co-location of batch instances through incremental scheduling [30], a misplaced instance with an inaccurate resource demand could be potentially rescheduled multiple times. Frequent rescheduling operations could result in low resource efficiency and application QoS degradation. Compared to AT, K8s reduces the rescheduling rate by 86%. Its vertical application autoscaling mechanism efficiently improves rescheduling efficiency and prevents multi-rescheduled cases. By contrast, CTCL reduces the rescheduling rate and cost in AT by 99% and 21% respectively in the best case (20c-24h). It manages to produce precise task co-location decisions through application-level workload prediction, while its rescheduling algorithm efficiently reduces the cost without multi-rescheduling. Under the same workload characterization class, the rescheduling rate is relatively lower under longer EOT, where behaviour identification is more accurate. On the other hand, class-20 ($k = 20$) enjoys the best performance among all the workload characterization classes under the same EOT. Overall, task rescheduling rate is regarded as one of the key metrics in evaluation of the accuracy of task co-location decisions. It shows how confident the algorithm is in resource usage prediction and performance interference management.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we propose the containerized task co-location (CTCL) scheduler for efficient task co-location of heterogeneous workloads under the context of a Cloud-based Cluster Management System. By employing an off-the-shelf algorithm such as K-means++ in workload characterization and behaviour identification, CTCL manages to make accurate task co-location decisions in an interference-aware manner. The experiments with Alibaba workload traces have shown that CTCL achieves significant resource utilization improvement and rescheduling rate reduction compared with the original traces and Kubernetes scheduling policies. Overall, our findings can be applied to large-scale cluster management for resource efficiency optimization, cost saving, and QoS assurance.

As future work, we plan to improve the scalability of our workload characterization approach under fast-growing workloads at an extreme scale through the dynamic incremental K-means++ clustering algorithm. Furthermore, an energy consumption model could be developed in CTCL for managing overall energy efficiency. Another possible direction would be implementing our scheduling policies in real-world CMS under the context of virtual cluster management, where the major concern is minimizing cloud resource rental costs through efficient task packing. Finally, we will extend CTCL to support task co-location under other workload management scenarios, such as network-intensive applications competing for shared network resources with potential performance interference.

ACKNOWLEDGMENTS

This work is supported by Melbourne Research Scholarship, China Scholarship Council, and Australia Research Council Discovery Project.

REFERENCES

[1] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai, "Imbalance in the cloud: An analysis on alibaba cluster trace," in *Proceedings of 2017 IEEE International Conference on Big Data*. IEEE, 2017, pp. 2884–2892.

[2] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of 2012 ACM Symposium on Cloud Computing*. ACM, 2012, pp. 7:1–7:13.

[3] H. Liu, "A measurement study of server utilization in public clouds," in *Proceedings of 2011 IEEE International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 2011, pp. 435–442.

[4] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *Proceedings of 2019 International Symposium on Quality of Service*. ACM, 2019, pp. 39:1–39:10.

[5] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of 2013 Annual Symposium on Cloud Computing*. ACM, 2013, pp. 5:1–5:16.

[6] N. Naik, "Building a virtual system of systems using docker swarm in multiple clouds," in *Proceedings of 2016 IEEE International Symposium on Systems Engineering*. IEEE, 2016, pp. 1–3.

[7] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of 2015 European Conference on Computer Systems*. ACM, 2015, p. 18.

[8] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of 2019 International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 107–120.

[9] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "Containercloudsim: An environment for modeling and simulation of containers in cloud data centers," *Software: Practice and Experience*, vol. 47, no. 4, pp. 505–521, 2017.

[10] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *Proceedings of 2015 International Conference on Advances in Computer Engineering and Applications*. IEEE, 2015, pp. 342–346.

[11] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace," in *Proceedings of 2018 ACM Symposium on Cloud Computing*. ACM, 2018, pp. 347–360.

[12] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.

[13] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of 2007 ACM-SIAM Symposium on Discrete algorithms*. SIAM Press, 2007, pp. 1027–1035.

[14] C. T. Joseph and K. Chandrasekaran, "Straddling the crevasse: A review of microservice software architecture foundations and recent advancements," *Software: Practice and Experience*, vol. 49, no. 10, pp. 1448–1484, 2019.

[15] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in *Proceedings of 2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 1287–1294.

[16] R. Birke, L. Y. Chen, and E. Smirni, "Data centers in the cloud: A large scale performance study," in *Proceedings of 2012 IEEE International Conference on Cloud Computing*. IEEE, 2012, pp. 336–343.

[17] I. Cano, S. Aiyar, and A. Krishnamurthy, "Characterizing private clouds: A large-scale empirical analysis of enterprise clusters," in *Proceedings of 2016 ACM Symposium on Cloud Computing*. ACM, 2016, pp. 29–41.

[18] V. Medel, O. Rana, J. Bañares, and U. Arronategui, "Modelling performance resource management in kubernetes," in *Proceedings of 2016 IEEE/ACM International Conference on Utility and Cloud Computing*. IEEE, 2016, pp. 257–262.

[19] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *Proceedings of 2010 International Conference on Network and Service Management*. IEEE, 2010, pp. 9–16.

[20] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 155–162, 2012.

[21] O. A. Abdul-Rahman and K. Aida, "Towards understanding the usage behavior of google cloud users: The mice and elephants phenomenon," in *Proceedings of 2014 IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2014, pp. 272–277.

[22] P. Garraghan, P. Townend, and J. Xu, "An analysis of the server characteristics and resource utilization in google cloud," in *Proceedings of 2013 IEEE International Conference on Cloud Engineering*. IEEE, 2013, pp. 124–131.

[23] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in google compute clusters," in *Proceedings of 2011 ACM Symposium on Cloud Computing*. ACM, 2011, pp. 3:1–3:14.

[24] M. E. Celebi, H. A. Kingravi, and P. A. Vela, "A comparative study of efficient initialization methods for the k-means clustering algorithm," *Expert systems with applications*, vol. 40, no. 1, pp. 200–210, 2013.

[25] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *Proceedings of the VLDB Endowment*, vol. 5, no. 7, pp. 622–633, 2012.

[26] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of 2011 USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2011, pp. 295–308.

[27] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of 2013 ACM European Conference on Computer Systems*. ACM, 2013, pp. 351–364.

[28] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *Proceedings of 2014 USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 285–300.

[29] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of 2013 International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2013, pp. 77–88.

[30] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: A fault-tolerant resource management and job scheduling system at internet scale," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1393–1404, 2014.

[31] C. Jiang, G. Han, J. Lin, G. Jia, W. Shi, and J. Wan, "Characteristics of co-allocated online services and batch jobs in internet data centers: A case study from alibaba cloud," *IEEE Access*, vol. 7, pp. 22 495–22 508, 2019.

[32] M. Xu, A. N. Toosi, and R. Buyya, "ibrownout: An integrated approach for managing energy and brownout in container-based clouds," *IEEE Transactions on Sustainable Computing*, vol. 4, no. 1, pp. 53–66, 2019.

[33] A. Nadjaran Toosi, C. Qu, M. D. de Assuno, and R. Buyya, "Renewable-aware geographical load balancing of web applications for sustainable data centers," *J. Netw. Comput. Appl.*, vol. 83, no. C, pp. 155–168, 2017.

[34] C. G. Kominos, N. Seyvet, and K. Vandikas, "Bare-metal, virtual machines and containers in openstack," in *Proceedings of 2017 Conference on Innovations in Clouds, Internet and Networks*. IEEE, 2017, pp. 36–43.

[35] Y. Omote, T. Shinagawa, and K. Kato, "Improving agility and elasticity in bare-metal clouds," in *Proceedings of 2015 International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015, pp. 145–159.

[36] A. Chung, J. W. Park, and G. R. Ganger, "Stratus: Cost-aware container scheduling in the public cloud," in *Proceedings of 2018 ACM Symposium on Cloud Computing*. ACM, 2018, pp. 121–134.

[37] Alibaba, "Alibaba Cluster Trace Program," https://github.com/alibaba/clusterdata/, 2018.

[38] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[39] Alibaba, "Alibaba Instance Families," https://www.alibabacloud.com/help/doc-detail/25378.htm, 2019.

[40] Kubernetes, "Kubernetes Vertical Pod Autoscaler," https://github.com/kubernetes/autoscaler/blob/master/vertical-pod-autoscaler/pkg/apis/autoscaling.k8s.io/v1beta2/types.go, 2019.