IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 23, 2019, accepted June 20, 2019, date of publication July 9, 2019, date of current version August 20, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2927279

Improving Resource Usages of Containers Through Auto-Tuning Container Resource Parameters

LIN CAI[1], YONG QI[1], WEI WEI[2], AND JINGWEI LI[1]
[1]School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China
[2]School of Computer Science and Engineering, Xi'an University of Technology, Xi'an 710048, China

Corresponding authors: Yong Qi (qiy@mail.xjtu.edu.cn) and Wei Wei (weiwei@xaut.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61672421, in part by the Key Research and Development Program of Shaanxi Province under Grant 2018ZDXM-GY-036, in part by the Scientific Research Program Funded by Shaanxi Provincial Education Department under Program 2013JK1139, and in part by the China Postdoctoral Science Foundation under Grant 2013M542370.

ABSTRACT Recently, the container-based virtualization has gained increasing attention and been widely used in cloud computing. In container products such as Docker, there are a number of parameters that can control container resource usages, to avoid the resource contention occurred when running too many containers concurrently. However, it is difficult to set parameter values accurately only based on experience while tuning the parameters manually is too time-consuming to be impractical. Therefore, it becomes a challenge to set appropriate resource parameter values automatically and quickly to optimize the resource usages of container. In this paper, we present an adaptive tuning framework, conTuner, to optimize the resource configuration of container online for a new application. conTuner contains two components: an optimized configuration pool that offers candidate resource configurations, as well as a configuration optimizer that gets the appropriate optimized configuration from the pool. We have deployed conTuner in a Docker cluster. The experimental results demonstrated that, for a new application, compared to the pre-set upper limit of container resource usages, the container performance is equal or better when using conTuner, and the set resource usage constraint is more accurate. Besides, conTuner can also forecast whether resource contention among multiple containers occurs before running them concurrently. The evaluation results indicate that the prediction accuracy is 87%.

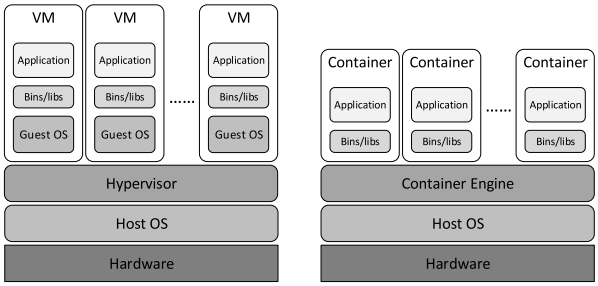
INDEX TERMS Cloud computing, Docker, online configuration optimization, parameter tuning, resource usage improving.

I. INTRODUCTION

With the continuous development and increasingly popularity of cloud computing, users can use the shared hardware and software resources provided by cloud services (e.g. IaaS, PaaS, SaaS) on demand, and pay only for the resources actually used. The academia has carried out extensive researches on various fields, such as performance optimization [1], [2], energy consumption [3], [4], and security [5], [6]. One of the core technologies in cloud computing is virtualization, which provides isolation on performance and resources. Traditional virtualization is implemented on the basis of hypervisors such as Xen [7] and KVM [8]. Recently, container technology, also known as operating system level virtualization, has attracted much attention due to its lightweight nature. It allows multiple isolated processes running in system to share a single operating system kernel, and leverage Linux cgroups to control the consumption of resources in the host. Currently, there are several commercial container implementations on the market, such as LXC (Linux Container) [9], Docker [10], rkt [11], and OpenVZ [12].

Fig. 1 represents the difference between the two virtualization technologies mentioned above. Hypervisor-based virtualization requires installing guest operating system for each virtual machine, while in container-based virtualization, all the virtualized instances (i.e. containers) run on the same

The associate editor coordinating the review of this manuscript and approving it for publication was Kaitai Liang.



(a) Hypervisor-based Virtualization (b) Container-based Virtualization

FIGURE 1. Comparison of hypervisor-based and container-based virtualization.

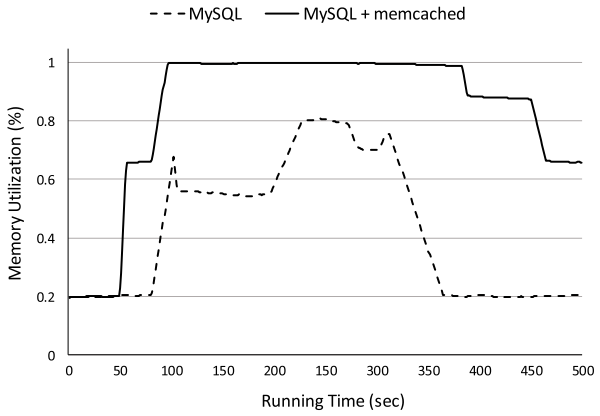


FIGURE 2. Resource contention among MySQL containers and Memcached containers.

operating system kernel shared by the host. Therefore, compared with hypervisor-based virtualization, container-based virtualization has several advantages. Firstly, the performance of container is better. In hypervisor-based virtualization, the hypervisor layer used for virtualizing hardware incurs significant performance overhead. Container-based virtualization eliminates this layer and thus can achieve near-native performance. Secondly, container-based virtualization does not need guest operating system, and is thus more lightweight and resource efficient. The startup speed of container is therefore an order of magnitude faster than virtual machine [13], which enables launching container on demand at runtime. Finally, due to the small resource requirement of a single container, the deployment density of containers on the same physical node is much higher as compared with virtual machines. For instance, [14] claims that, in the same hardware environment, the number of deployable containers can be four-to-six times more than that of virtual machines. With the above benefits, container-based virtualization, as a lightweight alternative to hypervisor-based virtualization, is being increasingly widely adopted in both development and production environments [15], [16].

On the other hand, unlike virtual machines that strictly isolate and pre-allocate resources, containers share resources such as CPU and memory on the host and compete for these shared resources at runtime. With the rising number of concurrently running containers, the increasing resource contention among containers could negatively affect applications running in the containers. Especially for containers running important and resource sensitive applications, resource contention from relatively less important containers would lead to a significant degradation in application performance or quality of service. To avoid such situation, commercial container-based virtualization products such as Docker provide some parameters [17] to limit the amount of resources that a single container can use. Optimizing the values of these resource parameters can help increase the overall resource utilization of system.

Figure 2 shows a case of contending resources among multiple containers in a Docker cluster. The dotted line in the figure indicates the cluster overall memory utilization when hosting 15 containers running application *MySQL*. In this case, the overall memory utilization is low at the beginning (the first 80 seconds), but then rises to a relatively high level. The solid line in the figure represents the cluster overall memory utilization when launching 10 new containers running application *Memcached* at 50 seconds after the *MySQL* containers startup. It can be seen that, when the memory usages of *MySQL* containers increase, the sum of the memory requirements of each container exceeds the amount of the host's physical memory, resulting in memory contention among containers, which eventually extends the running time taken by these containers. If one can predict the maximum memory requirements of applications *MySQL* and *Memcached* in advance and set the upper limits of memory usage for containers accordingly, the above situation can be avoided by, for example, migrating the *Memcached* containers to another host. Hence, through properly limiting the resource usages of containers, the amount of resources that need to be reserved for these containers can be calculated, which is helpful in arranging the type and number of containers running on the host reasonably.

However, since changing the parameter configuration of a running container is difficult, it is usually required to set the above container resource parameters empirically, that is, to pre-estimate the resource requirements of the application in the container according to certain characteristics (e.g. the size of dataset to process, the required calculation accuracy) of the application, and set the related parameters when launching containers. Unfortunately, such estimates are usually inaccurate, which would make a negative impact on application executions: overestimation would reduce the number of containers that can run concurrently, resulting in low utilization of host resources, while underestimation could lead to application performance degradation or even application failure. Due to the diversity of the resource usage patterns of applications, it is also difficult to establish a unified resource demand model to predict the resource usages of the application in container. On the other hand, since the number of container resource parameters is not a few (e.g. in Docker, there are more than 10 parameters for constraining container resource usages), manually tuning parameters

through trial-and-error is too time consuming that cannot be used in practice. Therefore, how to automatically and efficiently optimize the resource parameter configuration for a container has become an important issue in improving the resource utilization of the containerized cluster.

In this paper, we propose *conTuner*, a novel auto-tuning framework for container resource parameters, to address the aforementioned problem. conTuner is developed based on Docker and exploits a common observation that for applications with similar resource usage pattern, the influence of resource configuration change on their performance or resource utilization would also be similar. conTuner comprises two components: an optimized configuration pool, including a number of container resource configurations, each of which is optimized for a set of historical applications with a specific resource usage pattern, and a configuration optimizer, which chooses an appropriate optimized configuration from the optimized configuration pool to set resource parameters for container(s) running the new application. Since the configuration pool only needs to be constructed once offline, the time cost for tuning the container parameters online is small, which allows conTuner to optimize resource parameters for the container of the new application in a short time. We have deployed conTuner on an experimental cluster for evaluation. The experimental results demonstrate that, conTuner can improve the resource usages of a container without influencing the performance of application in the container, and predict the resource contention among containers in advance.

The main contributions of this paper are as follows:

- We propose an automatic tuning framework for resource parameters of container. This framework utilizes the historical applications grouped by the resource usage patterns to provide the new application with a number of optimized container resource configurations for selection. Experiment results show that, this framework can improve the container resource usages without impacting the container performance.
- We design a mechanism of building the optimized configuration pool from the historical applications. This mechanism can explore the resource usage patterns of an application from the multi-dimensional time series for classification, and use a two-phase random algorithm to accelerate searching for the optimized configuration.

The remainder of this paper is organized as follows: In section II we detail the architecture and design of conTuner. The implementation and evaluation results for conTuner are discussed in Section IV. Section V summarizes the related works and Section VI concludes this paper.

II. ARCHITECTURE AND DESIGN

The overall architecture of conTuner is shown in Figure 3. conTuner consists of two components: the optimized configuration pool and the configuration optimizer. The configurations in the optimized configuration pool are obtained

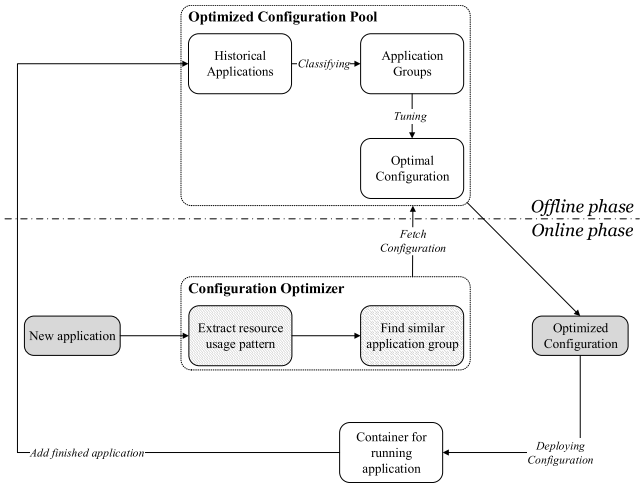


FIGURE 3. The architecture of conTuner.

through tuning parameters for the historical applications grouped by resource usage patterns respectively. This configuration pool is only constructed once in offline phase, and serves for optimizing the resource configuration of the new application in online phase. The configuration optimizer classifies the new application into one historical application group according to the resource usage patterns, and sets container resource parameters based on the corresponding configuration in the optimized configuration pool when launching containers of the new application.

In the rest of this section, we first outline the container parameters related to resource usages, and subsequently describe conTuner in detail.

III. CONTAINER RESOURCE PARAMETERS

In this subsection, we take Docker, one of the most popular commercial implementation of container, as an example to illustrate the container resource parameters. Docker exploits the cgroups in Linux kernel to manage the resources (e.g. CPU, memory and IO devices) used by containers, and provides a number of parameters to adjust the resource usages of container. Table 1 lists the parameters that have an obvious influence on the resource usages of containers.

However, determining the exact limits on container resource usages is not an easy task, it is thus necessary to pre-process these parameters to relieve the workload on parameter tuning. As shown in Table 1, according to different resources (CPU, memory and IO device), parameters in this table can be divided into three groups. The original tuning process can then be converted into three simpler processes: tuning CPU related, memory related and IO device related parameters respectively. Besides, as can be seen from Table 1, most parameters are either continuous or have a very large value range. We discretize the value ranges of the continuous parameters into a series of equidistant points. For example, we specify that for memory-related parameters, the interval between each two points is 256MB. Hence we can establish a discretized configuration grid $Grid\{c\}$ (c denotes an

TABLE 1. Container resource parameters.

Resource	*Name*	*Description*
CPU related	*-c, –cpu-shares*	CPU shares (relative weight).
	–cpu	Number of CPUs.
	–cpuset-cpus	CPUs in which to allow execution.
Memory related	*-m, –memory*	Memory limit.
	–memory-swap	Total memory limit (memory + swap), must be larger than *–memory*.
	–memory-swappiness	Tune a container's memory swappiness behavior.
	–memory-reservation	Memory soft limit, must be lower than *–memory*.
	–oom-kill-disable	Whether to disable OOM (out-of-memory) killer for the container or not.
IO device related	*–device-read-bps*	Limit read rate from a device.
	–device-write-bps	Limit write rate to a device.
	–device-read-iops	Limit read rate (IO per second) from a device.
	–device-write-iops	Limit write rate (IO per second) to a device.

available parameter configuration). The length unit of $Grid\{c\}$ is defined as the distance between two adjacent points in a dimension of $Grid\{c\}$, which is used to measure the Manhattan distance between any two configurations in $Grid\{c\}$.

A. CLUSTERING HISTORICAL APPLICATIONS

As mentioned earlier, conTuner classifies the historical applications into several groups according to resource usage patterns. In each group, applications exhibit similar performance behaviors and resource requirements under different resource conditions. In this subsection, we first collect relevant data of historical applications, then adopt a clustering method to group the historical applications according to the collected data.

For each historical application, we exploit a combination of open source tools *cAdvisor* and *influxDB* to collect performance and resource utilization data of this application as a basis for grouping. Specifically, during the running of a container, we collect the time series data of CPU usage, memory usage and disk usage, normalize each of them respectively, and integrate them as a three dimensional resource utilization signature belonging to that application. For a complex application that launches multiple containers to perform different functions respectively, in subsequent processing we take each of the containers as a separate ''small'' application. In this way, each historical application owns a unique resource utilization signature, which can be used to measure the similarity between applications on resource usages.

However, the lifespans of containers running different applications are often different, resulting in different lengths of the collected time series data. It is thus difficult to measure the difference between these unequal-length time series data directly using the traditional Euclidean distance metric. Such problem can be addressed by employing a Dynamic Time Wrapping (DTW) [18] based distance metric. For two time series with different lengths, DTW can ''wrap'' the time axis of one (or both) time series nonlinearly to find an optimal alignment between these two time series, and compute their similarities based on this alignment. The difference between the resource utilization signature of two applications can then be quantified as the distance in space through separately calculating the DTW distance of the time series of resource usages on each dimension. Calculating the distances between the resource utilization signatures of different applications respectively, the distribution of all the historical applications in a three-dimensional space can finally be obtained for subsequent clustering.

Next, conTuner employs a modified *K-medoids* based clustering approach to classify the historical applications into groups divided by resource usage patterns. *K-medoids* clustering algorithm is a variant of the classic *K-means* clustering algorithm. Compared with *K-means*, *K-medoids* selects actual data points to represent clusters, thus is more robust to noise and outliers. However, *K-medoids* still has the following shortcomings: Firstly, the accuracy of clustering is sensitive to the selection of initial cluster centers. Randomly generating the initial cluster centers might make the algorithm converge to local optimum. Secondly, the algorithm must explicitly specify the value of K (the number of clusters), but it is difficult to determine the appropriate K value before the algorithm begins. Finally, *K-medoids* needs to calculate the distance between data points repeatedly when updating cluster centers, leading to low efficiency of the algorithm. In allusion to these defects, conTuner has modified the *K-medoids* algorithm accordingly that exploits *Canopy* clustering to select the initial cluster centers. *Canopy* clustering algorithm [19] can fast and roughly divide a dataset into multiple partially overlapped subsets, which serve as the initial clusters for subsequent *K-medoids* clustering.

However, in *Canopy* clustering, the selection of the center point of each subset is still random. Here conTuner leverages the max-min criterion to improve the Canopy clustering algorithm so that the center points of subsets are dispersed as much as possible. Specifically, when creating a new subset, the distances between the candidate center points and the determined center points are firstly calculated. Then conTuner determines the shortest distance for each candidate point and selects the candidate point corresponding to the maximum of these shortest distances as the center point of the new subset. The details of the modified algorithm are described in Algorithm 1.

In the initialization phase (lines 1-3), the *Canopy* clustering's thresholds T_1 and T_2 need to be specified. We use cross-validation method to determine their values. The algorithm then exploits *Canopy* clustering (lines 4-25) to determine the number of subsets and their initial distributions. After the Canopy clustering completes, the algorithm assigns the data points belonging to multiple subsets to a subset contains the nearest central point (lines 26-31), thereby converting the partially overlapping subsets into disjoint clusters.

Algorithm 1 Modified K-Medoids Clustering Algorithm

Input: Dataset D

1: Set the distance thresholds T_1, T_2 ($T_1 > T_2$).
2: Initialize list *Medoids*, *nonMedoids* and *Candi*, two-dimensional list *Clusters*, upper triangular matrix *Distances*.
3: $nonMedoids \leftarrow D$
4: Randomly select p_1 from D.
5: Add a new list C_1 to *Clusters*. Insert p_1 to C_1 and *Medoids*.
6: Remove p_1 from D and *nonMedoids*.
7: $o_{cur} \leftarrow p_1$, $C_{cur} \leftarrow C_1$
8: **repeat**
9: **for** each point p in D **do**
10: **if** $distance(p, o_{cur}) < T_1$ **then**
11: Insert p to C_i.
12: **if** $distance(p, o_{cur}) < T_2$ **then**
13: Remove p from D.
14: **end if**
15: **else**
16: Insert p to *Candi* as a candidate point.
17: **end if**
18: **end for**
19: **if** *Candi* is not empty **then**
20: Find the next medoid p_i from *Candi* according to maximin criterion.
21: Add a new list C_i to *Clusters*. Insert p_i to C_i and *Medoids*.
22: Remove p from D and *nonMedoids*.
23: $o_{cur} \leftarrow p_i$, $C_{cur} \leftarrow C_i$
24: **end if**
25: **until** D is empty or *Candi* is empty
26: **for** each point n in *nonMedoids* **do**
27: **if** n belongs to more than one cluster in *Clusters* **then**
28: Find the medoids o_t nearest to n in *Medoids* and the corresponding cluster C_t.
29: Remove n from clusters other than C_t.
30: **end if**
31: **end for**
32: Calculate $S_{cur} = \sum_{j=1}^{k} \sum_{p_i \in C_j} distance(p_i, o_j)$, the cost of current clustering (k is the number of clusters).
33: **repeat**
34: **for** each point n in *nonMedoids* **do**
35: Fetch the medoid o' of the cluster C' that n belongs to.
36: Calculate S', the cost of clustering if swapping o' with n.
37: **if** $S' < S_{cur}$ **then**
38: Replace o' with n in *Medoids*.
39: Replace n with o' in *nonMedoids*.
40: Rebuild *Clusters*, assign each point in *nonMedoids* to the cluster with the nearest medoid.
41: Recalculate the cost of current clustering S_{cur}.
42: **end if**
43: **end for**
44: **until** the clusters no longer change

Output: $Clusters = \{C_1, C_2, \cdots\}$, *Medoids*

Finally, the *K-medoids* clustering (lines 32-44) is performed to iteratively update the clusters. In order to reduce the calculation amount, the algorithm only calculates the cost of exchanging data point inside the cluster, and do not exchange data points between different clusters. Moreover, since the process of calculating the distance between two data points (i.e., the resource utilization signatures of two historical applications) is complicated, an upper triangular matrix is used to record the distance between every two data points. For any two data points, if their distance is required at first time, the algorithm calculates the distance by comparing the similarity of the corresponding time series, and inserts the result into the matrix Distances. When the distance is required again, the corresponding value is fetched directly from the matrix. When the algorithm terminates, each obtained cluster corresponds to a group of historical applications with similar resource usage pattern, and the application corresponding to the cluster center is taken as the representative application of the group.

B. SEARCHING THE OPTIMAL CONFIGURATION

After grouping the historical applications, conTuner optimizes container resource configuration for applications in each group. Because the resource usage patterns of applications in the same group are similar, conTuner only needs to perform parameter tuning for the representative application of the group, and the obtained optimal container resource configuration can also be applied to all other applications in that group. In the previous section, we have listed several parameters that control the usage of container resources and pre-processed them to simplify the process of parameter tuning. Then conTuner is required to find the optimal configuration for each representative application in a discretized container resource configuration grid $Grid\{c\}$.

However, since it is too time-consuming to traverse all configurations in $Grid\{c\}$, a two-phase random search algorithm similar to [20] is introduced to improve search efficiency and avoid local optima. This algorithm divides the search process into two phases: global search and local search, as shown in Algorithm 2. The global phase is based on a modified genetic algorithm, aiming at discovering the local area to be explored further. In the local phase, the optimal configuration is found through hill climbing method. Furthermore, in order to speed up searching, the algorithm runs multiple containers concurrently in each iteration, each of which uses different resource configuration.

The inputs of the algorithm include: in global search, the population size n_1, the Manhattan distance between two configurations in the initial population r_1, the crossover probability P_c, the mutation probability P_m, the maximum iterative number N and the maximum times of consecutive global optimum unchanging b_0; in local search, the number of chosen configurations per iteration n_2 and the neighborhood radius r_2. n_1 and r_1 default to the average length of dimensions in $Grid\{c\}$, n_2 defaults to 2/3 of n_1, r_2 defaults to 1/5 of r_1, and the default value of P_c, P_m, N and b_0 is empirically set to 0.8, 0.05, 20, and 5, respectively.

After initialization, the algorithm enters in the global phase. Firstly, to generate the initial population of genetic algorithm, n_1 configurations are obtained using the latin hypercube sampling (LHS) method [21]. In order to avoid premature convergence, these configurations should be sufficiently dispersed: the Manhattan distance between each two configuration is specified to be not less than r_1. Otherwise

Algorithm 2 Two-Phase Random Search Algorithm

Input: in global search, the population size n_1, the distance between initial points r_1, the crossover probability P_c, the mutation probability P_m, the maximum iterative number N and the threshold of unchanging global optimum b_0; in local search, the number of chosen points per iteration n_2 and the neighborhood radius r_2.

$global\_search \leftarrow 0, no\_better \leftarrow 0$
repeat
  $Conf\_Set \leftarrow$ LHS_Sampling (n_1)
until $\forall c_1, c_2 \in Conf\_Set, distance_{c_1,c_2} >= r_1$
$c_{candi} \leftarrow$ FindBest($Conf\_Set$), $c_{cur} \leftarrow c_{candi}$
Sort $Conf\_Set$ from good to bad, obtain $Conf\_List$.
while $global\_search < N$ or $no\_better < b_0$ **do**
  Calculate the probability that each configuration in $Conf\_List$ is chosen.
  Perform selection operation, obtain $Conf\_sel$. $Conf\_sel$ must include c_{cur}.
  Select pairwise configurations for crossover from $Conf\_sel$ by crossover probability P_c.
  Perform crossover operation, obtain $Conf\_cross\_tmp$.
  $c_{candi\_c} \leftarrow$ FindBest($Conf\_cross\_tmp$), accept configuration based on Metropolis rule, obtain $Conf\_cross$.
  Select configurations for mutation from $Conf\_cross$ by mutation probability P_m.
  Perform mutation operation, obtain $Conf\_mut\_tmp$.
  $c_{candi} \leftarrow$ FindBest($Conf\_mut\_tmp$), accept configuration based on Metropolis rule, obtain $Conf\_mut$.
  if c_{candi} is not better than c_{candi_c} **then**
    $c_{candi} \leftarrow c_{candi\_c}$, replace the worst configuration in $Conf\_mut$ with c_{candi}.
  end if
  if c_{candi} is better than c_{cur} **then**
    $c_{cur} \leftarrow c_{candi}, no\_better \leftarrow 0$
  else
    no_better++, Add c_{cur} to $Conf\_mut$.
  end if
  if the number of configurations in $Conf\_mut$ is less than n_1 **then**
    Complement $Conf\_mut$ using the top ranked configurations in $Conf\_sel$.
  end if
  Sort $Conf\_mut$ from good to bad, obtain a new $Conf\_List$.
  $global\_search$++
end while
$c_{candi} \leftarrow c_{cur}$
while $global\_search > 0$ **do**
  $N_{cur} \leftarrow$ GetAllNeighbors(c_{cur}, r_2)
  repeat
    $c_{cur} \leftarrow c_{candi}$
    $Conf\_Set \leftarrow$ SelectNeighbors $(N_{cur}, n_2, c_{cur}, 1)$
    $c_{candi} \leftarrow$ FindBest($Conf\_Set$)
  until c_{candi} is not better than c_{cur}
  repeat
    $Conf\_Set \leftarrow$ LHS_Sampling (n_1)
  until $\forall c_1, c_2 \in \{Conf\_Set \cup \{c_{cur}\}\}, distance_{c_1,c_2} >= r_1$
  $c_{candi} \leftarrow$ FindBest $(Conf\_Set)$
  if c_{candi} is better than c_{cur} **then**
    $c_{cur} \leftarrow c_{candi}$
  else
    $global\_search \leftarrow 0$
  end if
end while

Output: c_{cur}

the algorithm would invoke LHS repeatedly until acquiring a set of satisfied configurations. Then the algorithm launches n_1 containers and sets resource parameters of each container using one of the configurations obtained before. These containers are sorted by performance after they finish running. If there are multiple containers having the equal performance, their order would be decided by the resource utilization during run time. According to the sorting result, the algorithm selects some of the configurations from the current population with a certain probability. Note that the current best configuration must be included in the selected population $Conf\_sel$. Next, the algorithm selects configurations from the population $Conf\_sel$ by crossover probability to perform crossover operation, then launches a new batch of containers to evaluate the obtained child configurations. Referring the idea of simulated annealing, the child configurations that meet the Metropolis rule are preserved in population $Conf\_cross$. Similarly, the algorithm performs mutation operation and stores the accepted child configuration in population $Conf\_mut$, and takes actions to guarantee that, from $Conf\_cross$ to $Conf\_mut$, the best configuration in population would not worsen. After that, the algorithm updates the global best configuration. If the number of configurations in population $Conf\_mut$ is insufficient, the algorithm selects the first k configurations of $Conf\_sel$ to complement $Conf\_mut$ (k is the number of configurations need to be added). The complemented $Conf\_mut$ is treated as the population for next iteration. The above procedure repeats until the iteration number reaching the threshold N, or the global best configuration does not change after consecutive b_0 iterations. The algorithm then switches to the local search phase to pinpoint the local optimal configuration in a neighborhood with radius r_2.

In the local search phase, in each iteration the algorithm randomly selects n_2 configurations with a Manhattan distance of 1 from the current central configuration, and determines the best performing configuration through actual running. This iterative process finishes when it is impossible to find a configuration better than the current central configuration. The final central configuration is then taken as the local optimal configuration.

Finally, the algorithm enters the global phase again to check whether the current local optimal configuration is global optimum. If finding a better configuration, the algorithm would make a local search once again around the newly found configuration, otherwise the algorithm terminates and outputs the optimal configuration.

Using the above two-phase random search algorithm, conTuner can determine the optimal container resource configuration for each representative historical application, and build the optimized configuration pool based on these resource configurations. However, the resource requirements of some applications are sensitive to certain characteristics (e.g. data scale, calculation accuracy). When such characteristics change, the original optimal resource configuration

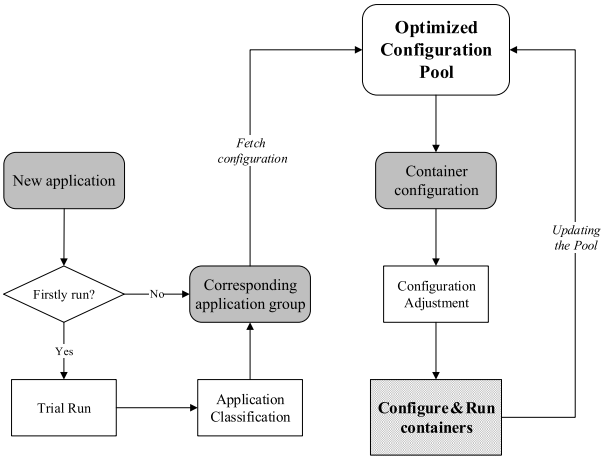


FIGURE 4. The Workflow of Configuration Optimizer.

might be no longer optimum even for the same application. Hence, for a specific application, it is necessary to identify its sensitive characteristics and analyze the correlation between these characteristics and the resource demands of the application. conTuner has proposed a configuration adjustment scheme for each representative application in order to optimize container resource parameters properly when the application's sensitive characteristics change.

C. THE CONFIGURATION OPTIMIZER

After building the optimized configuration pool offline, conTuner starts up the configuration optimizer to optimize container resource parameters online for new applications. If a new application has already run before, the optimizer takes the configuration corresponding to the historical application group it belongs to directly from the optimized configuration pool. Otherwise the optimizer would first classify the new application into one of the established historical application groups based on the resource usage pattern, then employ the corresponding configuration in the configuration pool to set resource parameters for the container running this application, and record the resource usages while the container is running. Figure 4 shows the workflow of the configuration optimizer.

However, the resource usage pattern of an application never run before is unknown until it actually runs, making it difficult to classify the application directly. conTuner thus conducts a short trial run for the new application before actual running. The trial run only takes very little time by adopting a variety of time-saving ways (e.g. sampling large dataset or relaxing high precision requirement).

During the trial run, the configuration optimizer collects resource usage data from the launched container to generate a three dimensional resource utilization signature of the new application. Then the optimizer compares this signature with the resource utilization signature of every representative historical application. Because the resource utilization signatures are multidimensional time series that are difficult to compare, directly the above comparisons are all conducted based on the DTW distance metric. After finding out the representative application whose resource utilization signature is nearest to the new application, the optimizer fetches the corresponding optimized resource configuration from the configuration pool.

As mentioned in the previous section, since the new application and the historical applications are often different in certain sensitive characteristics (e.g. data scale, calculation accuracy), the optimized resource configuration obtained from the configuration pool might not be fully suitable for the new application. The configuration optimizer thus employs the previously described corresponding configuration adjustment scheme to adjust the obtained resource configuration, then uses the adjusted configuration to set resource parameters for the container launched to run the new application.

On the other hand, as the historical applications continually increases, it is necessary to timely update the grouping of historical applications. However, as described earlier, reclassifying historical applications, especially re-searching the optimal resource configuration for each application group, takes a considerable amount of time. Hence the frequency of grouping update should not be too high. It is specifies here that, the process of updating the grouping begins if the number of accumulated historical applications reaches the preset threshold, or if the resource utilization or performance of the container using the recommended resource configuration differs greatly from expectation (e.g. greater than 10%).

Finally, we analyze the time cost of the configuration optimizer. Since the configuration optimizer performs a series of operations before the container actually runs, the time spent on these operations should be taken into consideration in the calculation. In these operations, apart from the trial run of the application that takes some time, the time spent on other operations (for example, sampling the dataset or adjusting the already obtained configuration) is very little and can be ignored. Therefore, the total time cost of the configuration optimizer can be approximately expressed as:

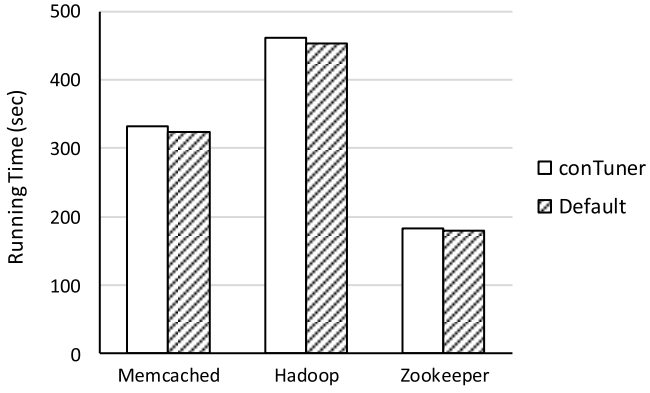
$$\begin{aligned} T_{total} &= T_{trial} + T_{optimal} \\ &= \alpha \cdot T_{original} + T_{optimal} \end{aligned} \tag{1}$$

where α indicates the reduction degree in the running time of the trial run relative to the original running time. $T_{original}$ and $T_{optimal}$ respectively represent the running time of containers whose resource parameters are set using the original configuration and the optimal configuration. It is obvious that when α is small enough, the configuration optimizer can optimize container resource configuration with a small extra time cost.

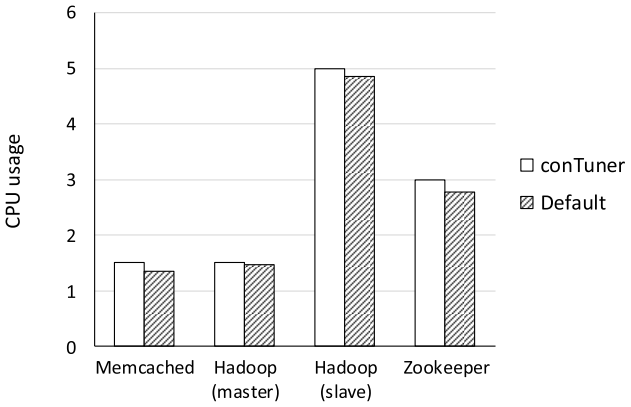
IV. EVALUATION

A. EXPERIMENTAL SETTING

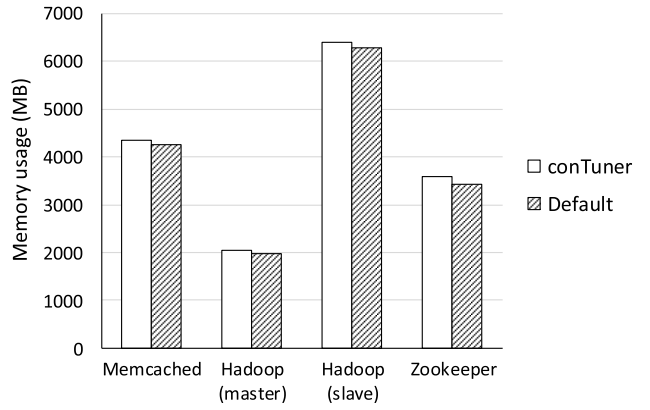
We have implemented and deployed conTuner on our experimental cluster. The cluster consists of two blade servers, each of which includes two Xeon E5649 2.53GHz hexa-core processors, 64 GB memory, and 8TB hard disk. We use



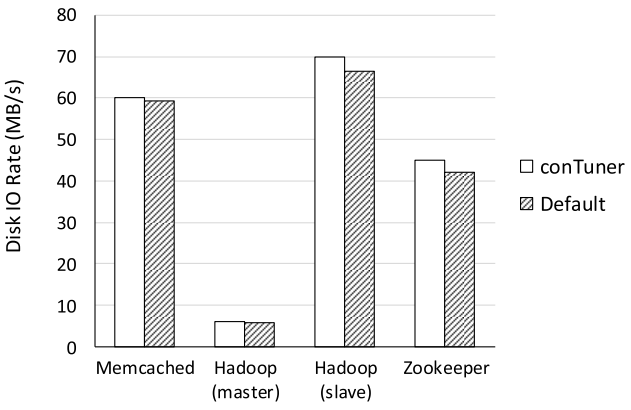
(a) Container performance.



(b) CPU usage limit (conTuner) vs. Maximum CPU usage (Default).



(c) Memory usage limit (conTuner) vs. Maximum Memory usage (Default).



(d) Disk usage limit (conTuner) vs. Maximum Disk usage (Default).

FIGURE 5. The accuracies of the resource configuration optimized by conTuner compared with no constraints on resource usage.

Ubuntu 14.04.5 64-bit with Linux kernel 4.4.0 and Docker 17.05.0.

In the evaluation, we select 10 commonly used applications representing various types to construct workloads, as shown in Table 2. We take the images of these applications from Docker Hub [22] and build corresponding container(s). For each application, a continuous operation (e.g. insert millions of record into MySQL, or perform Terasort test on Hadoop) is performed throughout the lifespan of container(s). Note that for big data processing frameworks Hadoop and Spark, the container for master node and the container for slave node are treated as two different types of containers because their resource usage patterns are different.

B. EVALUATION RESULTS

Firstly, we evaluate the accuracy of the configuration obtained by tuning with conTuner, namely, compare the limit of resource usages in the optimized configuration and the actual resource requirements of the application, which is acquired when running the application in container(s) without resource usage constraint. Specifically, we select six odd-numbered applications from Table 2 as the historical applications of conTuner for training, and select three applications in Table 2: Memcached, Hadoop and Zookeeper as

TABLE 2. Applications for evaluation.

No.	*Application*	*Type*
1	Floating Point Calculation	Linpack
2	MySQL	Database System
3	MongoDB	
4	Redis	Caching System
5	Memcached	
6	Tomcat	Web Service
7	HAProxy	
8	TensorFlow	Deep Learning Framework
9	Caffe	
10	Hadoop (master, slave)	Big Data Processing Framework
11	Spark (master, slave)	
12	Zookeeper	Coordination System

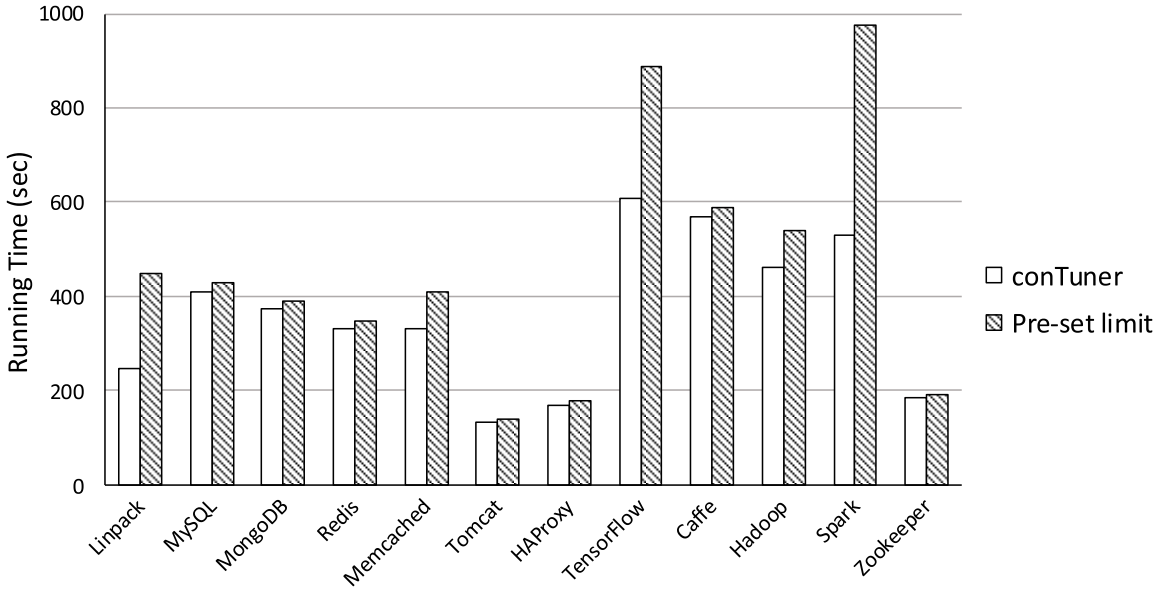


FIGURE 6. The performance of containers optimized by conTuner and containers using pre-set usage limit.

new incoming applications to test conTuner. In the tests, each application runs respectively in container(s) that use conTuner to optimize resource parameters and in container(s) that does not limit resource usages (the default situation). The container performances in above two cases are shown in Figure 5(a), while Figure 5(b) to 5(d) respectively compare the upper bounds on CPU/memory/disk usage of container in the optimized resource configurations with the maximum usage of the corresponding resource for the container without resource limit. As can be seen from the figures, the difference in the running time of container is within 5% and the maximum gap of resources is less than 10%. Since the performance of container without resource usage limit would not be worse than the container with resource usage limit, it can be conclude that conTuner can predict the actual resource requirements of application accurately, and generate a near-optimal container resource configuration based on the predicted result. Although the container performance under the optimized resource configuration is slightly worse than the default situation, setting the container resource usage limit is beneficial to reserve the resources required for running container accurately.

Next, we compare the influence on the running results for using conTuner to customize container resource usage limit and using the pre-set resource limit of container. Here the pre-set resource limit refers to the resource combination which is set according to the host's hardware configuration, and is unrelated to the resource requirements of the specific application. Specifically, in this experiment, we leverage the models trained in the previous experiment to optimize the resource configuration for each of the applications in Table 2 and specifies the upper limit of resources can be used by container at runtime. On the other hand, we use trial-and-error method to empirically select the most appropriate one for each application from the following five resource combinations as the resource usage cap of container: 1. (1 CPU core and 2GB memory), 2. (2 CPU core and 2GB RAM), 3. (2 CPU core and 4GB RAM), 4. (2 CPU core and 8GB memory) and 5. (4 CPU core and 8GB memory). The comparison results are shown in Figure 6 and 7.

Figure 6 represents the container performance for each application when setting different resource usage caps. It can be observed that, for more than one-third of the applications, the container performance when using the pre-set resource usage limit is obviously lower than when using conTuner for optimization. For the remain applications, the container performances in this two cases are close, but as illustrated in Figure 7, when using the pre-set resource usage limit, their actual maximum CPU or memory usages have not reached the pre-set values. In other words, for most applications, the pre-set resource usage ceilings are either too low or too high. Hence it can be said that the container resource configuration optimized by conTuner is better than the pre-set container resource usage limit, which cannot match the container resource requirement well without prior knowledge of the application.

Finally, we validate the ability of conTuner to predict the resource contention among containers. We concurrently run a group of applications that are randomly selected from Table 2 in a single server. These applications run twice: once in containers that have no resource usage constraint and once in containers that use conTuner to optimize resource parameters. In the second case, if the sum of the usage caps of all containers on a specific resource exceeds the actual capacity of this resource in the server, the excessive container(s) are placed on the other server. The above procedure is repeated

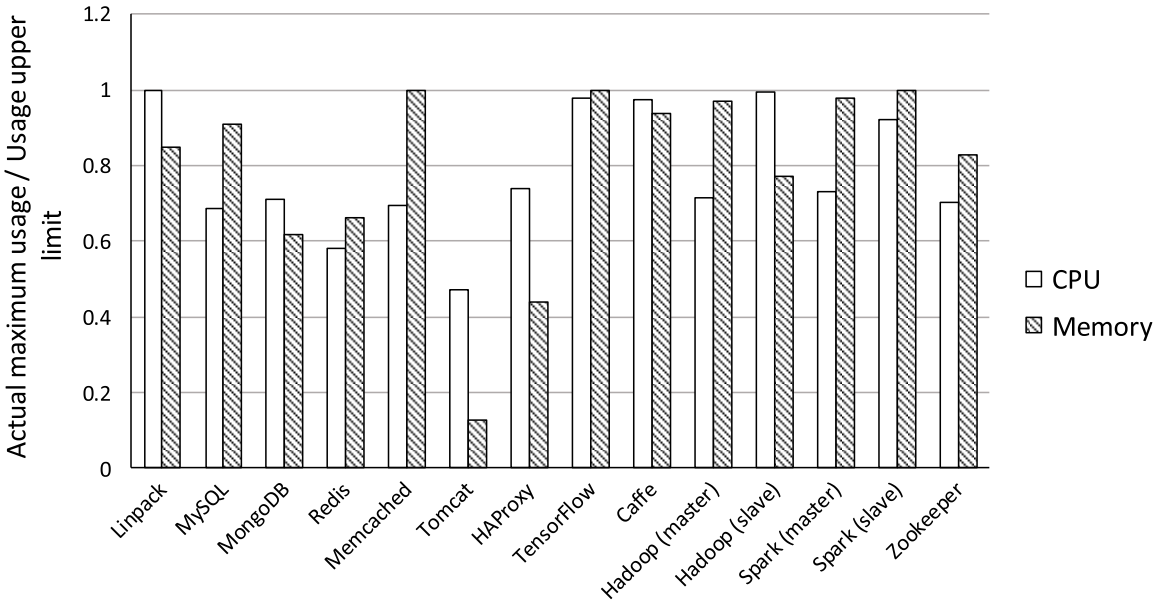


FIGURE 7. The ratio of the actual maximum CPU/memory usage to the upper limit when using pre-set resource limit.

200 times and each time the selected applications are different. Table 3 lists the statistics for the results of the above experiments. From the table we can see that, when running in containers without resource usage constraint, the resource contention among containers is detected for nearly three quarters of the application groups. If using conTuner to optimize container resource configuration before these application group actually running, the resource contentions of more than 85% application groups can be predicted. For another quarter of application groups, in most cases, the difference between the container performances when using conTuner and when do not limit the container resource usages is within 10%, namely, the resource configuration optimized by conTuner canfit the real resource demands. Overall, 174 of the 200 experiments meet expectation (detecting resource contention or the optimized configuration performs well), which means that the forecast accuracy of conTuner reaches 87%.

TABLE 3. Results after running 200 groups of applications.

Without resource usage constraint	*Apply conTuner for optimization*
Detect resource contention: **167**	Put excessive containers to another server: **145**
	Run all containers on original server: **22**
Running normally: **33**	Put containers to another server: **3**
	The performance difference is within 5%: **23**
	The performance difference is between 5% and 10%: **6**
	The performance difference is over 10%: **1**
Total application groups: **200**	

V. RELATED WORKS

The performance issues of containers has been highly concerned. Some of the recent works discussed the resource optimization for containerized platforms.

Reference [23] investigated the influence of different container resource configurations on the performance of Spark jobs running in Docker containers, and proposed a performance prediction model based on support vector regression (SVR). In [24], an engine based on a modified k-nearest neighbor algorithm was designed to recommend resource configuration automatically for Hadoop workloads running on container-driven clouds. Reference [25] adopted several machine learning techniques, such as linear regression (LR), support vector machine (SVM) and artificial neural network (ANN), to model the relationship between application performance and the resource parameter configuration for Docker containers, and assessed the accuracy of the established performance model. However, these approaches only aim at a specific application (Spark or Hadoop), and are not suitable for complex application environments. Reference [26] presented a flexible container-based tuning system that enables users to find appropriate network topologies and routing algorithms for big data applications. In [27] a resource allocation approach based on stable matching theory is proposed to optimize the resource utilization of the container cloud while improving application performance. Reference [28] employed an ant colony optimization (ACO)-based algorithm to optimize the distribution and scheduling of Docker containers. Reference [29] developed a linear programming based framework that can optimize resource allocation for application-oriented Docker containers and

support automatic resource expansion when demand changes. Reference [30] proposed a resource allocation mechanism for container cloud based on combined double auction and simulated annealing algorithms.

In addition, there are also numerous works focusing on tuning the parameters of big data platforms. Reference [31] introduced a concept of elastic container to Hadoop YARN that can optimize parameter configuration and resource allocation dynamically based on the collected real-time statistics. In [32] a method combining the binary classification and multi-classification is adopted to tune the parameters of Spark automatically. Reference [33] proposed a gray-box performance model for Spark that predict the performance of each stage using regression algorithms. However, none of the above approaches involves containers and the resource used by containers.

VI. CONCLUSION

The container-based virtualization has been widely adopted in the field of cloud computing. In order to avoid the resource contention among multiple containers running concurrently, several container resource parameters have been provided to control the resource usages of container. However, for a specific application, it is difficult to determine appropriate parameter values just relying on experience, whereas the time cost of manual parameter tuning is too high. In this paper, we propose an adaptive tuning framework, conTuner, to optimize container resource parameters for new applications online. conTuner first build an optimized configuration pool offline, where each configuration is optimized for a group of historical applications classified by resource usage pattern. Then conTuner uses a configuration optimizer to obtain the corresponding optimized configuration from the configuration pool according to the resource usage pattern of the new application. We deployed conTuner in an experimental cluster and the evaluation results show that, compared with using the pre-set resource usage limit of container, using the container resource configuration optimized by conTuner can reflect the actual resource requirements of the container more accurately while achieving equal or better container performance. In addition, when predicting the resource contention among containers running concurrently, the accuracy of conTuner reaches 87%.

In the future, we plan to extend our work in two aspects: first, we will add optimization for network IO related parameters and second, we will put the weight factors of applications into account when optimizing the container resource configuration.

ACKNOWLEDGMENT

The authors are very grateful to anonymous reviewers for their constructive comments.

REFERENCES

[1] P. Zheng, Y. Qi, Y. Zhou, P. Chen, J. Zhan, M. R.-T. Lyu, "An automatic framework for detecting and characterizing performance degradation of software systems," *IEEE Trans. Rel.*, vol. 63, no. 4, pp. 927–943, Dec. 2014.

[2] P. Chen, Y. Qi, X. Li, D. Hou, and M. R.-T. Lyu, "ARF-predictor: Effective prediction of aging-related failure using entropy," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 4, pp. 675–693, Jul./Aug. 2016.

[3] P. Wang, Y. Qi, and X. Liu, "Power-aware optimization for heterogeneous multi-tier clusters," *J. Parallel Distrib. Comput.*, vol. 74, no. 1, pp. 2005–2015, 2014.

[4] H. Dou, Y. Qi, W. Wei, and H. Song, "A two-time-scale load balancing framework for minimizing electricity bills of Internet data centers," *Pers. Ubiquitous Comput.*, vol. 20, no. 5, pp. 681–693, 2016.

[5] S. Qi, Y. Zheng, M. Li, L. Lu, and Y. Liu, "Secure and private RFID-enabled third-party supply chain systems," *IEEE Trans. Comput.*, vol. 65, no. 11, pp. 3413–3426, Nov. 2016.

[6] S. Qi and Y. Zheng, "Crypt-DAC: Cryptographically enforced dynamic access control in the cloud," *IEEE Trans. Dependable Secure Comput.*, to be published.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Dec. 2003.

[8] A. K. Qumranet, Y. K. Qumranet, D. L. Qumranet, U. L. Qumranet, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, vol. 1, 2007, pp. 225–230.

[9] M. Helsley, "LXC: Linux container tools," in *IBM DevloperWorks Technical Library*, vol. 11, 2009.

[10] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.

[11] *RKT*. Accessed: Jun. 20, 2019. [Online]. Available: https://coreos.com/rkt/

[12] *Openvz*. Accessed: Jun. 20, 2019. [Online]. Available: https://openvz.org/

[13] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Adv. Sci. Technol. Lett.*, vol. 66, nos. 105–111, p. 2, 2014.

[14] S. J. Vaughan-Nichols. (2014). *What is Docker and Why is it So Darn Popular?* [Online]. Available: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/

[15] Datadog. (2018). *8 Surprising Facts About Real Docker Adoption*. [Online]. Available: http://www.datadoghq.com/docker-adoption/

[16] RightScale. (2018). *Rightscale 2018 State of the Cloud Report*. [Online]. Available: https://www.rightscale.com/2018-cloud-report

[17] *Docker Documentation*. Accessed: Jun. 20, 2019. [Online]. Available: https://docs.docker.com

[18] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowl. Inf. Syst.*, vol. 7, no. 3, pp. 358–386, 2005.

[19] A. Mccallum, K. Nigam, and L. H. Ungar, "Efficient clustering of high-dimensional data sets with application to reference matching," in *Proc. 6th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2000, pp. 169–178.

[20] L. Cai, Y. Qi, W. Wei, J. Wu, and J. Li, "mrMoulder: A recommendation-based adaptive parameter tuning approach for big data processing platform," *Future Gener. Comput. Syst.*, vol. 93, pp. 570–582, Apr. 2019.

[21] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *Proc. 13th Int. Conf. World Wide Web*, 2004, pp. 287–296.

[22] *Docker Hub*. Accessed: Jun. 20, 2019. [Online]. Available: https://hub.docker.com

[23] K. Ye and Y. Ji, "Performance tuning and modeling for big data applications in docker containers," in *Proc. Int. Conf. Netw., Archit., Storage (NAS)*, Aug. 2017, pp. 1–6.

[24] R. Zhang, M. Li, and D. Hildebrand, "Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Mar. 2015, pp. 365–368.

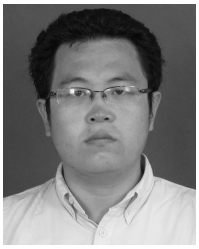
[25] K. Ye, Y. Kou, C. Lu, Y. Wang, and C.-Z. Xu, "Modeling application performance in docker containers using machine learning techniques," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2018, pp. 1–6.

[26] Y. Yu, H. Zou, W. Tang, L. Liu, and F. Teng, "Flex tuner: A flexible container-based tuning system for cloud applications," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Mar. 2015, pp. 145–154.

[27] X. Xu, H. Yu, and X. Pei, "A novel resource scheduling approach in container based clouds," in *Proc. IEEE 17th Int. Conf. Comput. Sci. Eng.*, Dec. 2014, pp. 257–264.

[28] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *Proc. 9th Int. Conf. Knowl. Smart Technol. (KST)*, Feb. 2017, pp. 254–259.

[29] X. Guan, X. Wan, B.-Y. Choi, S. Song, and J. Zhu, "Application oriented dynamic resource allocation for data centers using docker containers," *IEEE Commun. Lett.*, vol. 21, no. 3, pp. 504–507, Mar. 2017.
[30] C. Chen, Z. Zhang, and X. Xie, "Container cloud resource allocation based on combinatorial double auction," in *Proc. 3rd Int. Conf. Intell. Inf. Process.*, 2018, pp. 146–151.
[31] X. Ding, Y. Liu, and D. Qian, "JellyFish: Online performance tuning with adaptive configuration and elastic container in hadoop yarn," in *Proc. IEEE 21st Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2015, pp. 831–836.
[32] G. Wang, J. Xu, and B. He, "A novel method for tuning configuration parameters of spark based on machine learning," in *Proc. IEEE 18th Int. Conf. High Perform. Comput. Commun., IEEE 14th Int. Conf. Smart City, IEEE 2nd Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Dec. 2016, pp. 586–593.
[33] Z. Chao, S. Shi, H. Gao, J. Luo, and H. Wang, "A gray-box performance model for Apache Spark," *Future Gener. Comput. Syst.*, vol. 89, pp. 58–67, Dec. 2018.
[34] W. Wei and Y. Qi, "Information potential fields navigation in wireless ad-hoc sensor networks," *Sensors*, vol. 11, no. 5, pp. 4794–4807, 2011.
[35] W. Wei, X.-L. Yang, P.-Y. Shen, and B. Zhou, "Holes detection in anisotropic sensornets: Topological methods," *Int. J. Distrib. Sensor Netw.*, vol. 8, no. 10, pp. 135054-1–135054-9, 2012.
[36] S. Qi, Y. Zheng, M. Li, Y. Liu, and J. Qiu, "Scalable industry data access control in RFID-enabled supply chain," *IEEE/ACM Trans. Netw.*, vol. 24, no. 6, pp. 3551–3564, Dec. 2016.
[37] Q. Ke, J. Zhang, H. Song, and Y. Wan, "Big data analytics enabled by feature extraction based on partial independence," *Neurocomputing*, vol. 288, pp. 3–10, May 2018.



LIN CAI received the B.Eng. degree in computer science from Zhejiang University, in 2006. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Xi'an Jiaotong University, China. His research interests include cloud computing, big data analytics, and performance optimization.

YONG QI received the Ph.D. degree from Xi'an Jiaotong University, China, where he is currently a Full Professor with the Department of Computer Science and Technology. His research interests include operating systems, distributed systems, cloud computing and big data system, and system security and application. He is a member of ACM.

WEI WEI received the M.S. and Ph.D. degrees from Xi'an Jiaotong University, China, in 2005 and 2011, respectively, where he is currently an Associate Professor with the School of Computer Science and Engineering. His research interests include wireless networks, wireless sensor networks application, image processing, mobile computing, distributed computing, pervasive computing, the Internet of Things, sensor data clouds, and so on. He is a Senior Member of CCF. He ran many funded research projects as a Principal Investigator and a Technical Member.

JINGWEI LI received the B.S. and M.S. degrees in computer science from Lanzhou University, China, in 2010 and 2013, respectively. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Xi'an Jiaotong University, China. Her research interests include performance optimization, big data analytics, and efficient resource management.