Communication-Aware Container Placement and Reassignment in Large-Scale Internet Data Centers

Liang Lv, Yuchao Zhang, Yusen Li, Ke Xu, *Senior Member, IEEE*, Dan Wang, *Senior Member, IEEE*, Wendong Wang, *Member, IEEE*, Minghui Li, Xuan Cao, and Qingqing Liang

***Abstract*— Containerization has been used in many applications for isolation purposes due to its lightweight, scalable, and highly portable properties. However, to apply containerization in large-scale Internet data centers faces a big challenge. Services in data centers are always instantiated as a group of containers, which often generate heavy communication workloads and therefore resulting in inefficient communications and downgraded service performance. Although assigning the containers of the same service to the same server can reduce the communication overhead, this may cause heavily imbalanced resource utilization since containers of the same service are usually intensive to the same resource. To reduce communication cost as well as balance the resource utilization in large-scale data centers, we further explore the container distribution issues in a real industrial environment and find that such conflict lies in two phases—container placement and container reassignment. The objective of this paper is to address the container distribution problem in these two phases. For the container placement problem, we propose an efficient communication aware worst fit decreasing algorithm to place a set of new containers into data centers. For the container reassignment problem, we propose a two-stage algorithm called Sweep&Search to optimize a given initial distribution of containers by migrating containers among servers. We implement the proposed algorithms in Baidu's data centers and conduct extensive evaluations. Compared with the state-of-the-art strategies, the evaluation results show that our algorithms perform better up to 70% and increase the overall service throughput up to 90% simultaneously.**

***Index Terms*— Container communication, multi-resource load balance, large-scale data centers, container placement, container reassignment.**

Manuscript received April 29, 2018; revised January 6, 2019; accepted January 11, 2019. Date of publication January 31, 2019; date of current version February 14, 2019. The work of L. Lv was supported by the National Key Research and Development Program of China under Grant 2018YFB0803405. The work of Y. Zhang was supported in part by the China Postdoctoral Science Foundation under Grant 2018M630117, in part by the National Natural Science Foundation of China under Grant 61802024, and in part by the Huawei Autonomous and Service 2.0 Project under Grant A2018185. The work of Y. Li was supported in part by the Baidu Songguo Plan, in part by NSF of China under Grant 61602266, and in part by NSF of Tianjin under Grant 16JCYBJC41900. The work of K. Xu was supported in part by the National Key Research and Development Program of China under Grant 2018YFB0803405, in part by the China National Funds for Distinguished Young Scientists under Grant 61825204, and in part by the Beijing Outstanding Young Scientist Project. The work of D. Wang was supported in part by PolyU G-YBAG. *(Liang Lv and Yuchao Zhang contributed equally to this work.) (Corresponding authors: Yusen Li; Ke Xu.)*

L. Lv and K. Xu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: lvl16@mails.tsinghua.edu.cn; xuke@tsinghua.edu.cn).

Y. Zhang is with the School of Software Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China (e-mail: yczhang@bupt.edu.cn).

Y. Li is with the Department of Computer Science, Nankai University, Tianjin 300071, China (e-mail: liyusen@nbjl.nankai.edu.cn).

D. Wang is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong (e-mail: dan.wang@polyu.edu.hk).

W. Wang is with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China (e-mail: wdwang@bupt.edu.cn).

M. Li, X. Cao, and Q. Liang are with Baidu, Beijing 100094, China (e-mail: liminghui@baidu.com; caoxuan@baidu.com; liangqingqing@baidu.com).

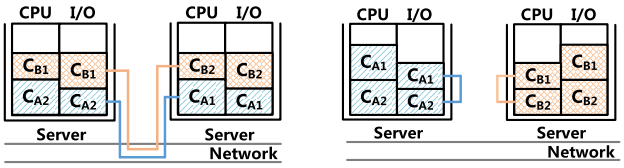
Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2019.2895473

I. Introduction

CONTAINERIZATION [1] has become a popular virtualization technology due to many promising properties such as lightweight, scalable, highly portable and good isolation, and the emergence of software containerization tools, e.g., docker [2], further allows users to create containers easily on top of any infrastructure. Therefore, more and more Internet service providers are deploying their services in the form of containers in modern data centers.

Generally, each Internet service has several modules which are instantiated as a set of containers, and the containers belonging to the same service often need to communicate with each other to deliver the desired service [3]–[6], resulting in heavy cross-server communications and downgrading service performance [3], [7]. If these containers are placed on the same server, the communication cost can be greatly reduced. However, the containers belonging to the same service are generally intensive to the same resource (e.g., containers of the big data analytics services [8]–[10] are usually CPU-intensive, and containers of the data transfer applications [11]–[15] are usually network I/O-intensive). Assigning these containers on the same server may cause heavily imbalanced resource utilization of servers, which could affect the system availability, response time and throughput [16], [17]. First, it prevents any single server from getting overloaded or breaking down, which improves service availability. Second, servers usually generate exponential response time when the resource utilization is high [18], load balancing guarantees acceptable resource utilizations for servers, so that the servers can have fast response time. Third, no server will be a bottleneck under balanced workload, which improves the overall throughput of the system.



(a) High network communication overhead, balanced resource utilization.

(b) Imbalanced resource utilization, low network communication overhead.

Fig. 1. The conflict between container communication and server resource utilization.

TABLE I
RESOURCE UTILIZATION IN A DATA CENTER FROM BAIDU[1]

Resource	Top 1%	Top 5%	Top 10%	Mean
CPU	0.943	0.865	0.821	0.552
MEM	0.979	0.928	0.890	0.626
SSD	0.961	0.927	0.875	0.530

Figure 1 shows an example. Suppose there are two services (denoted by S_A and S_B) to be deployed on two servers. Each service has two containers (C_{A1}, C_{A2} and C_{B1}, C_{B2}, respectively). The containers of S_A are CPU-intensive while the containers of S_B are network I/O-intensive. Figure 1(a) shows a solution which assigns one of S_A's containers and one of S_B's containers on each server. This approach achieves high resource utilization on both CPU and network I/O, but incurs high communication cost between the two servers. Figure 1(b) shows another solution where the containers of the same service are assigned on the same server. The communication overhead is thus significantly reduced, however, the utilization of CPU and network I/O is highly imbalanced on the two servers.

We further explore the conflict between container communication and resource utilization in a data center with 5,876 servers from Baidu. According to our knowledge, the containers of the same service in this data center are placed as close as possible in order to reduce the communication cost. Table I gives the top 1%, top 5% and top 10% CPU, MEM (Memory) and SSD (Solid State Drives) utilization of servers in this data center, which shows that the utilization of resources is highly imbalanced among servers.

Reducing container communication cost while keeping balanced server resource utilization is never an easy problem. In this paper, we try to address such conflict in large-scale data centers. Specifically, such conflict lies in two related phases of an Internet service's life cycle, i.e., container placement and container reassignment, and we accordingly study two problems. The first is Container Placement Problem, which strives to place a set of newly instantiated containers into a data center. The objective of this phase is to balance resource utilization while minimizing the communication cost of these containers after placement. The second is Container Reassignment Problem, which tries to optimize a given placement of containers by migrating containers among servers. Such reassignment approach can be used for online periodical adjustment of the placement of containers in a data center. We formulate these two problems as multi-objective optimization problems, which are both NP hard.

For the Container Placement Problem, we propose an efficient Communication Aware Worst Fit Decreasing (CA-WFD) algorithm, which subtly extends the classical Worst Fit Decreasing bin packing algorithm to container placement. For the Container Reassignment Problem, we propose a two-stage algorithm named Sweep&Search which can seek a container migration plan efficiently. We deploy our algorithms in Baidu's data centers and conduct extensive experiments to evaluate the performance. The results show that the proposed algorithms can effectively reduce the communication cost while simultaneously balancing the resource utilization among servers in real systems. The results also show that our algorithms outperform the state-of-the-art strategies up to 90% used by some top containerization service providers.

This paper is extended from [19] with significant improvements including:

- We disclose a new problem (i.e., the Container Placement Problem) that places a set of newly instantiated containers into a data center, which is a necessary and important phase in services' life cycle.
- We propose the CA-WFD algorithm to solve the Container Placement Problem and conduct extensive experiments to evaluate the performance.
- We refine the algorithms proposed for the Container Reassignment Problem, and significantly extend the experimental study for this problem.

The rest of this paper is structured as follows. Section II introduces the architecture of container group based services. Definitions of Container Placement Problem and Container Reassignment Problem are given in Section III. Our solutions to the two problems are proposed in Sections IV and V, respectively. Section VI compares our solutions with state-of-the-art designs by extensive evaluations. We implement our solutions in large-scale data centers of Baidu, and the details are given in Section VII. Section VIII covers related work. At last, Section IX concludes the paper.

II. CONTAINER GROUP BASED ARCHITECTURE

Containerization is gaining tremendous popularity recently because of its convenience and good performance on deploying applications and services. First, containers provide good isolations with namespace technologies (e.g., chroot [20]), eliminating conflicts with other containers. Second, containers put everything in one package (code, runtime, system tools, system libraries) and do not need any external dependencies to run processes [21], making containers highly portable and fast to distribute.

To ensure service integrity, a function of a particular application may instantiate multiple containers. For example, in Hadoop, each mapper or reducer should be implemented as one container, and the layers in a web service (e.g., load balancer, web search, backend database) are deployed as container groups. These container groups are deployed in

[1]http://www.baidu.com

cloud or data centers, and managed by orchestrators such as Kubernetes [22] and Mesos [23]. Using name services, these orchestrators can quickly locate the containers on different servers, so application upgrade and failure recovery can be well handled. As containers are easy to build, replace or delete, such architecture makes it convenient to maintain container group-based applications.

But the container group based architecture also introduces side effects, i.e., the low communication efficiency. As the functions deployed in the same container group belong to the same service, they need to exchange control messages and transfer data. Therefore, communication efficiency within a container group greatly affects the overall service performance [3]. However, simple consolidation strategies may result in imbalanced utilization of multiple resources, because containers of the same group are usually intensive to the same resource. The above orchestrators provide the possibility to leverage containers, but how to manage container groups for reducing communication overhead and balancing resource utilization is still a pending problem.

III. Problem Definition

In this section, we go into the above trade-off by analyzing the overall costs under specified constraints. Let $\mathbf{H}$ denote the set of servers in a data center. Each server has multiple types of resources. Let $\mathbf{R}$ denote the set of resource types. For each server $h \in \mathbf{H}$, let $P(h,r)$ denote the capacity of resource $r \in \mathbf{R}$. Let $\mathbf{S}$ denote the set of services. Each service is built in a set of containers. Each container may have several replicas. For a specific container c, let D_c^r denote its resource requirement for resource r ($r \in \mathbf{R}$). The set of containers to be placed is denoted by $\mathbf{C}$.

A. Objective

According to Section II, there are two aspects when quantifying the total overhead of any distribution status, i.e., the communication cost and the resource utilization (we consider both in-use resources and residual resources).

1) Communication Cost: So far we can formulate the overall communication cost as follows: for each container $c \in \mathbf{C}$, let $H(c)$ denote the server that container c assigned to. For a pair of containers c_i and c_j, let $f(c_i, c_j)$ denote the communication cost incurred by these two containers. Since the communication overhead exists mainly in host networks, if c_i and c_j are placed on the same server ($H(c_i) = H(c_j)$), the communication cost is negligible, i.e., $f(c_i, c_j) = 0$. Thus, the overall communication cost for the data center is the sum of the communication cost produced by all possible container pairs, which is given by

$$C_{cost} = \sum_{\forall c_i, c_j \in \mathbf{C}, c_i \neq c_j} f(H(c_i), H(c_j)). \tag{1}$$

The next two metrics measure the resource utilizations of servers, which are *Resource Utilization Cost* and *Residual Resource Balance Cost*.

2) Resource Utilization Cost: If the resource utilization of a server is much higher than others, it will easily become the bottleneck of a service, seriously degrading the overall performance. The ideal situation is that all servers enjoy equal resource utilization. For each resource type $r \in R$, the resource utilization cost for r is defined as the variance of resource usage for r of all servers, i.e.,

$$\sum_{h \in \mathbf{H}} \frac{[U(h,r) - \bar{U}(r)]^2}{|\mathbf{H}|}, \tag{2}$$

where $U(h,r)$ denotes the utilization of resource r on server h, $\bar{U}(r)$ is the mean utilization of resource r of all servers and $|\mathbf{H}|$ is the number of servers. This metric can reflect whether resource r is used in a balanced way among servers. The total resource utilization cost for the data center is the sum of the resource utilization cost of all resource types, which is given by

$$U_{cost} = \sum_{r \in R} \sum_{h \in \mathbf{H}} \frac{[U(h,r) - \bar{U}(r)]^2}{|\mathbf{H}|}, \tag{3}$$

3) Residual Resource Balance Cost: Any amount of CPU resource without any available RAM is useless for coming requests, so the residual amount of multiple resources should be balanced [16]. For two different resources r_i and r_j, let $t(r_i, r_j)$ represent the target proportion between resource r_i and resource r_j. The residual resource balance cost incurred by r_i and r_j is defined as

$$cost(r_i, r_j) = \sum_{h \in \mathbf{H}} max\{0, A(h,r_i) - A(h,r_j) \times t(r_i, r_j)\} \tag{4}$$

where $A(h,r)$ refers to the residual available resource r on server h. This metric can reflect whether different types of resources are used according to the expected proportion. The total residual resource balance cost for the data center is the sum of the residual balance cost of all possible pairs of resource types, which is given by

$$B_{cost} = \sum_{\forall r_i, r_j \in \mathbf{R}, r_i \neq r_j} cost(r_i, r_j) \tag{5}$$

Based on the above definitions, the overall resource utilization of servers can be measured by the sum of the total resource utilization cost and the total residual resource balance cost. It is easy to see that smaller cost indicates more balanced resource utilization among servers.

A commonly used approach to optimize multiple optimization objectives is to transfer multiple objectives into a single scalar [24]. We adopt this approach in this paper and define the objective to be minimized as a weighted sum of all the costs defined above, i.e.,

$$Cost = w_U * U_{cost} + w_B * B_{cost} + w_C * C_{cost}. \tag{6}$$

B. Constraints

To minimize the above cost, containers should be placed or reassigned to the most suitable servers, but this process should satisfy some strict constraints.

1) Capacity Constraint: First, the resource consumed by the containers on each server cannot exceed the capacity of the server for each resource type, i.e.,

$$\sum_{c\in\mathbf{C},H(c)=h} D_c^r \le P(h,r), \quad \forall h\in\mathbf{H},\ \forall r\in\mathbf{R}. \tag{7}$$

2) Conflict Constraint: Second, as mentioned earlier, each container may have several replicas for parallel processing purpose. Generally, the replicas of the same container cannot be placed on the same server. Let $\Gamma(c,c')$ denote whether c and c' are the replicas of the same container, with $\Gamma(c,c')=1$ indicating yes and $\Gamma(c,c')=0$ otherwise. The Conflict Constraint can be represented by

$$\Gamma(c,c')=1 \Rightarrow H(c)\neq H(c'), \quad \forall c,c'\in\mathbf{C},\ c\neq c'. \tag{8}$$

3) Spread Constraint: Third, a specific function in a high performance application is usually implemented on multiple containers to support concurrent operations. For example, a basic search function in web services is usually instantiated in different servers or even different data centers. As these containers are sensitive to the same resource, they cannot be put on the same server. Otherwise, there will be serious waste of other resources like memory and I/O. Therefore, for each service $S_i\in\mathbf{S}$, let $M(S_i)\in\mathbf{N}$ be the minimum number of different servers where at least one container of S_i should run, we can define the following Spread Constraint for each service:

$$\sum_{h_i\in\mathbf{H}} min(1,|c\in S_i\in\mathbf{S}|H(c)=h_i|)\ge M(S_i), \quad \forall S_i\in\mathbf{S}. \tag{9}$$

4) Co-Locate Constraint: Fourth, some services require critical data transmission delay among containers. In order to satisfy the latency requirement, the containers with critical frequent interactions should be assigned to the same server. Let $\Lambda(c,c')$ denote whether c and c' should be co-located on the same server, with $\Lambda(c,c')=1$ indicating yes and $\Lambda(c,c')=0$ otherwise. The Co-locate Constraint can be represented by

$$\Lambda(c,c')=1 \Rightarrow H(c)=H(c'), \quad \forall c,c'\in\mathbf{C},\ c\neq c'. \tag{10}$$

5) Transient Constraint: The Container Reassignment problem assumes a given placement of containers and tries to further improve the initial placement by migrating containers among servers. For each container $c\in\mathbf{C}$, let $H(c)$ and $H'(c)$ denote the original server and the new server (after migration) that container c is assigned to. In order to guarantee service availability, any migrated container cannot be destroyed at the original server until the new instance is created on the new server. Therefore, the resources are consumed at both original server $H(c)$ and the new server $H'(c)$ during container migration. This constraint can be represented by

$$\sum_{c\in\mathbf{C},H(c)=h\bigcup H'(c)=h} D_c^r \le P(h,r), \quad \forall h\in\mathbf{H},\ \forall r\in\mathbf{R}. \tag{11}$$

Based on the above discussions, we formally define the problems to be addressed in this paper.

- **Container Placement Problem (CPP)**. Given a set of new containers, to find the optimal placement of containers such that the total cost defined by (6) is minimized while the constraints (7), (8), (9) and (10) are not violated.
- **Container Reassignment Problem (CRP)**. Given an initial placement of containers, to find the optimal new placement of containers such that the total cost defined by (6) is minimized while the constraints (7), (8), (9), (10) and (11) are not violated.

IV. Container Placement Problem

In this section, we firstly show that CPP is NP-hard and then propose a heuristic algorithm called Communication Aware Worst Fit Decreasing to approximate the optimal solution of CPP.

A. Problem Analysis

To prove that CPP is NP-hard, let us consider the Multi-Resource Generalized Assignment Problem [25] (MRGAP) first. Given m agents $\{A=1,2,...,m\}$, n tasks $\{T=1,2,...,n\}$ and l resources $\{R=1,2,...,l\}$, each agent i has $cap_{i,r}$ units of resource r and each task j requires $req_{j,r}$ units of resource r. Assigning a task j to agent i induces a cost $cost_{i,j}$. MRGAP tries to assign each task to exactly one agent with the purpose of minimizing the total cost without violating the resource constraints, i.e.,

$$\begin{aligned}&\min \sum_{i\in T} cost_{i,A(i)}\\ &\text{s.t.} \sum_{j\in A,i\in T(j)} req_{i,j,r}\le cap_{j,r}, \quad \forall r\in R\end{aligned} \tag{12}$$

where $A(i)$ denotes the agent task i is assigned to, and $T(j)$ denotes the set of tasks on agent j.

It is widely accepted that the MRGAP is a strongly NP-hard problem [26]. We note that MRGAP is essentially a simplified version of CPP. Suppose we deploy service S into some empty servers with only the capacity constraint considered (i.e., we do not consider constraints (8), (9) and (10)). Because the total cost is 0 before deploying S (note that the servers are initially empty), we can denote the final cost by $\sum_{c\in\mathbf{C}_S}\Delta Cost_c$, where $\mathbf{C}_S$ denotes the set of containers of service S, and $\Delta Cost_c$ denotes the increment in $Cost$ in Equation (6) by placing container c. This Simplified CPP (SCPP) can be formulated as follows:

$$\begin{aligned}&\min \sum_{c\in\mathbf{C}_S}\Delta Cost_c\\ &\text{s.t.} \sum_{c\in\mathbf{C_S},H(c)=h} D_c^r\le P(h,r), \quad \forall h\in\mathbf{H},\ \forall r\in\mathbf{R}.\end{aligned} \tag{13}$$

From Equations (12) and (13), it is easy to see that MRGAP is equivalent to SCPP if we regard agents as servers and tasks as containers. In other words, MRGAP is a special case of CPP. Therefore, it follows that CPP is NP-hard.

Although MRGAP is a special case of CPP, there are key differences between them, which make existing solutions to MRGAP inapplicable for CPP. Firstly, there are more

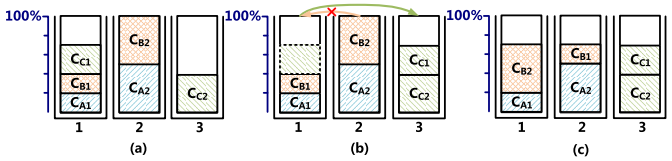


Fig. 2. There are three services each with two containers: (a) the current assignment; (b) the impractical reassignment; (c) the optimal solution.

constraints in CPP, thus feasible solutions to MRGAP may be infeasible to CPP. Secondly, unlike in MRGAP, the assignment cost in CPP (i.e., increment of $Cost$ induced by an assignment) is dependent upon assignment sequence, which could even be negative (note that a proper placement can improve resource utilization without increasing communication overhead and thus decreases the overall cost).

B. CA-WFD Algorithm

As large-scale data centers usually have thousands of containers and servers, the approaches that try to find optimal solutions are impractical for CPP due to the high computation complexity. In this section, we propose a heuristic algorithm to approximate the optimal solution to CPP, which is extended from the Worst Fit Decreasing (WFD) [27] strategy.

The basic idea of WFD is to sort the items in a decreasing order according to their sizes and each item is assigned to the bin with largest residual capacity. WFD is widely used for load balancing [28] because WFD tends to distribute slack among multiple bins. However, we face several challenges to apply WFD in CPP. The first is how to measure the sizes of containers and the capacities of servers. A commonly used approach is to transfer the multi-dimensional resource vector into a scalar. Since different designs of the scaler may yield different performances [29], we need to carefully scalerize the resource vectors in CPP. Moreover, WFD is traditionally applied for balancing resource utilization, so we have to extend WFD to CPP, where both resource load balance and communication overhead reduction are considered.

To measure the sizes of containers, we define the *Dominant Requirement* of a container, i.e., the maximum requirement on different resources, which is expressed as $\max_{r\in\mathbf{R}} D_c^r$. We use the *Weighted Sum* of residual resources to measure the available capacity of a server, which is defined as $\sum_{r\in R} w_r * A(h,r)$, where w_r refers to the weight of resource r. In fact, motivated by prior researches [29], we proposed and tested several designs based on real-world environments and finally choose the two metrics.

To extend WFD to CPP, instead of simply picking the server with the largest free space in WFD, we take two steps to select a server for a new container. In the first step, we put emphasis on load balance, where d candidate servers with the most available resources are selected. In the second step, to reduce communication overhead, we choose the server with the most containers that belongs to the same service with the new container.

We propose the Communication Aware Worst Fit Decreasing (CA-WFD) algorithm shown in Algorithm 1.

Algorithm 1 CA-WFD

1: $\mathbf{C} \leftarrow$ the set of new containers to be placed
2: $\mathbf{H} \leftarrow$ the set of servers
3: Sort containers in $\mathbf{C}$ according to their sizes
4: **while** $\mathbf{C} \neq \emptyset$ **do**
5: Pick the container $c \in \mathbf{C}$ with the largest size
6: Pick d servers $\mathbf{H}_d$ with the largest available capacity that can accommodate c without violating any constraint
7: Pick the server $h \in \mathbf{H}_d$ that accommodates the most containers that belongs to the same service with c
8: Assign c to h
9: $\mathbf{C} \leftarrow \mathbf{C} \backslash \{c\}$
10: **end while**

The algorithm sorts the containers according to their sizes (measured by *Dominant Requirements*) in line 3. Then, it repeatedly assigns the largest container until all the containers are assigned (lines 4-10). Each time the algorithm picks the top d servers with the largest residual capacity (measured by *Weighted Sum* of residual resources) and chooses the one with the most containers that belongs to the same service with c to minimize the increment of C_{cost}.

V. Container Reassignment Problem

CRP aims to optimize a given initial placement of containers by migrating containers among servers. As mentioned earlier, all the constraints should be satisfied during the migrations in order to guarantee the online services. Since containers are already placed on servers initially, the residual capacities of servers that can be utilized during migrations are quite limited, making CRP challenging. Classical heuristic algorithms have been used to solve similar problems [30]–[32]. However, the existing approaches are inefficient to handle big containers at the hot hosts due to transient constraints in CRP.

A. Problem Analysis

Figure 2 shows the additional constraint and challenge in CRP, where single resource and homogeneous servers are considered for simplicity. There are six containers which are placed on three servers. Initially (as shown in Figure 2(a)), three containers (C_{A1}, C_{B1} and C_{C1}) are placed on Server 1, whose resource requirements are 20%, 20% and 30%, respectively. Two containers (C_{A2} and C_{B2}) are placed on Server 2, whose resource requirements are both 50%. One container (C_{C2}) is placed on Server 3, whose resource requirement is 40%. Suppose the capacity of each server is 100%. Obviously, the optimal placement of containers is as shown in Figure 2(c), i.e., each server has two containers and the total resource utilization is 70%. As illustrated in Figure 2(b), it is impossible to get the optimal placement from the initial placement by migrating containers concurrently. This is because in order to achieve the optimal placement, we need to move C_{C1} from Server 1 to Server 3, and move C_{B2} from Server 2 to Server 1. However, migrating C_{B2} from Server 2 to Server 1 is infeasible since the transient constraint will be violated on

Server 1 (the sum of resource consumption on Server 1 will exceed its capacity if C_{B2} is migrated).

For the above example, if we first move C_{C1} from Server 1 to Server 3 and suppose the transient resource is released on Server 1 after migration, then C_{B2} can be migrated from Server 2 to Server 1 without violating any constraint. Inspired by this observation, we propose a two-stage container reassignment algorithm named *Sweep&Search* to solve the problem.

B. Sweep&Search Algorithm

The Sweep&Search algorithm has two stages, which are *Sweep* and *Search*. The *Sweep* stage tries to handle the large containers on the hot servers, i.e., trying to migrate the large containers to the expected locations. Based on the placement produced by the *Sweep* stage, the *Search* stage adopts a tailored variable neighborhood local search to further optimize the placement of containers.

Note that, the Sweep&Search algorithm is only used to compute the migration plan, i.e., which server each container will be migrated to. So, all the placement changes (e.g., migrate, shift, swap) in the algorithm description are hypothetical. After the migration plan is figured out, the containers are physically "migrated" to their target servers as follows: first, for each container, a new replica of the container is constructed in the target server; second, the workload mapped to the "old" replica of the container is redirected to the new replica; third, the old replica of the container is physically deleted. The Sweep&Search algorithm has taken the transient resource constraints into account when computing the migration plan, so the resource constraints at both the original servers and the target servers can be always satisfied during migration.

1) Sweep: Recall that one of our objectives is to balance resource utilization among servers. The traditional approaches for load balancing normally move workload directly from servers with high resource utilization to servers with low resource utilization. However, as shown in Figure 2, the large containers are hard to migrate due to the transient resource constraints. To address this issue, we propose a novel two step approach in the *Sweep* stage. In the first step, we try to empty the spare servers as much as possible by moving out containers from the spare servers to other servers. This will free up space for accommodating more large containers from the hot servers. In the second step, we move large containers from hot servers to spare servers so that resource utilization among servers can be balanced.

The pseudo-code of *Sweep* is shown in Algorithm 2. The algorithm first selects a set of hot servers (i.e., the servers whose resource utilization is higher than a predefined safety threshold). Suppose the number of hot servers is N. The algorithm then tries to clear up N spare servers (i.e., the top N servers with the lowest resource utilization). When working on a spare server, the algorithm tries to migrate as many containers as possible from the spare server to other servers (lines 5-8). For a specific container c, the procedure *FindHost* returns a normal server (i.e., neither a hot server nor a spare server) that can accommodate c. After that, the resources

Algorithm 2 Sweep

```
1: Sort H in descending order according to the residual capacity
2: H_hot ← {h|U(h) > safety threshold}
3: N ← the size of H_hot
4: H_spare ← top N spare servers
5: for each container c on h ∈ H_spare do
6:   h' ← FindHost(c)
7:   Migrate c from h to h'
8: end for
9: for each h ∈ H_hot do
10:   while U(h) > Ū(h) do
11:     pick a container c on h
12:     pick a spare server h' ∈ H_spare that can accommodate
        c without violating any constraint
13:     Migrate c from h to h'
14:   end while
15: end for
```

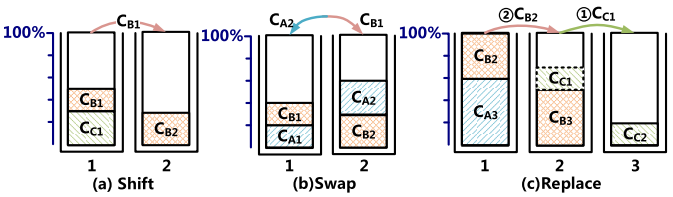


Fig. 3. Three kinds of moves Sweep&Search explores.

occupied by the containers which have been migrated can be released on the spare servers. Then, the algorithm tries to migrate containers from hot servers to spare servers to balance the resource utilization (lines 9-15). Specifically, the algorithm iterates over the hot servers and repeatedly migrates containers from each hot server to spare servers until the resource utilization of the hot server is below the average of all servers if possible.

2) Search: The *Sweep* stage mainly focuses on balancing the resource utilization among servers. However, the communication cost may still be high after the *Sweep* stage. The *Search* stage will further optimize the solution produced by the *Sweep* stage using a local search algorithm. The local search algorithm incrementally adjust the placement through three basic movements: shift, swap and replace.

A shift move is to reassign a container from one server to another server (Figure 3(a)). It is the most simple neighbor exploration that directly reduces the overall cost. For example, reassigning a container from a hot server to a spare server will reduce U_{cost}; moving a CPU-intensive container from a server with little residual CPU will reduce B_{cost}; moving a container nearer to its group members reduces C_{cost}.

A swap move is to exchange the assignment of two containers on two different servers (Figure 3(b)). It is easy to see that the size of the swap neighborhood is $O(n^2)$, where n is the number of containers. To limit the branch number, we cut off the neighbors that obviously violate the constraints and worsen the overall cost.

Algorithm 3 Search

$P_{crt} \leftarrow$ the initial container placement
repeat
  Sort $\mathbf{H}$ by resource utilization
  $N \leftarrow \mathbf{H}(Top(\delta)) + \mathbf{H}(Tail(\delta))$
  $P_{shift} \leftarrow$ shiftSearch(P_{crt}, N)
  $P_{swap} \leftarrow$ swapSearch(P_{crt}, N)
  $P_{replace} \leftarrow$ replaceSearch(P_{crt}, N)
  $P_{crt} \leftarrow \arg\min(Cost(P)), P \in \{P_{shift}, P_{swap}, P_{replace}\}$
until $Cost(P_{crt}) < T$
$P_{best} \leftarrow P_{crt}$
Output P_{best}

The replace move is more complex than the shift move and the swap move, which is to shift a container from one server (the original server) to another server (the relay server) and meanwhile shifts zombie containers on the relay server to other servers (the target servers) (Figure 3(c)). A zombie container is a container that was planned to move here (from other servers) in the earlier search stage but the actual operation has not been executed yet. We represent zombie containers with dashed edges in the figure. Since the zombie containers have not been migrated, it will not incur additional overhead if we reassign them to other servers.

It is obvious that replace is more powerful than shift and swap, but the overhead is much higher. This is because there are so many potential movements for a zombie container and *replace* should explore all the possible branches. Fortunately, the overhead can be bounded. In each iteration of the *Search* algorithm, one *shift* and one *swap* will be accepted, which will generate 3 zombies. So there are 3 cases for the next *replace* phase. In each branch, we try to move the zombie container away from the assigned host, and this is another *shift* operation. So in the ith iteration, there are at most $3i$ zombie containers. If we assume the overhead of exploring a *shift* neighbor is o_s, the total overhead of Sweep&Search is linear to o_s.

The local search algorithm is described by Algorithm 3. The algorithm repeats iteratively until the overall cost falls to the pre-set threshold T. In each iteration, three procedures are executed, namely *shiftSearch*, *swapSearch* and *replaceSearch*.

The *shiftSearch* procedure attempts to migrate containers from hot servers to spare servers to reduce the total cost. It first randomly selects a set of hot servers and a set of spare servers. Then, it tries to shift a container on the selected hot servers to one of the spare servers with the condition that the total cost is reduced after the shift move.

The *swapSearch* procedure aims at reducing the total cost through swapping the locations of containers. It first randomly selects a set of hot servers and a set of non-hot servers. For each container on the selected hot servers, the algorithm tries to find a container on the selected non-hot servers such that the total cost is reduced if the locations of the two containers are swapped.

TABLE II
SERVER CONFIGURATION

Data Center	Server No.	CPU	MEM	SSD
DC_A	2077	1.000	1.000	0.400
	169	0.342	0.750	1.000
	112	0.589	0.750	1.000
DC_B	2265	0.830	0.667	0.242
	1394	0.430	0.333	0.242
	160	1.000	1.000	0.606

TABLE III
SERVICE INFORMATION

Data Centers	Algorithms	U_{cost}	B_{cost}	C_{cost}
DC_A	CA-WFD	0.069	0.703	0.403
	Random	0.114	0.824	0.400
	HA	0.111	0.709	0.424
	ENF	0.087	0.715	0.406
	Binpack	0.163	0.686	0.396
DC_B	CA-WFD	0.048	0.053	0.695
	Random	0.062	0.147	0.747
	HA	0.050	0.116	0.769
	ENF	0.042	0.090	0.758
	Binpack	0.088	0.170	0.698

The *replaceSearch* procedure tries to reduce the total cost by reassigning a container from h_A to h_B under the premise to move a zombie container from h_B to h_C. It firstly chooses a set of hot servers as origin servers. For each container c on the origin server h, it selects a set of non-hot servers as relay servers. For each zombie container c' on the relay server h', *replaceSearch* tries to find a target server h'' from a set of randomly selected spare servers, such that the overall cost is reduced if reassigning c from h to h' meanwhile moving c' from h' to h''.

In Appendix A, we give a detailed algorithm analysis and prove that the deviation between Sweep&Search and the theoretical optimal solution has an upper bound.

VI. EVALUATION

We have conducted extensive experiments with variant parameter settings in Baidu's large-scale data centers to evaluate CA-WFD and Sweep&Search. As we cannot deploy comparison systems in real data centers due to safety concerns, evaluations are performed in two experimental data centers, where there are 2,513 servers accommodating 10,000+ containers of 25 services in DC_A and 4,361 servers accommodating 25,000+ containers of 29 services in DC_B.

The configurations of servers in the two data centers are summarized in Table II. Resource requirements of typical services in DC_A (S_{Ai}) and DC_B (S_{Bi}) are given in Table III. The values in Table II and III are after normalization, where the top server configuration of each dimension of resource is normalized to 1. These real-world data show that both the server configurations and resource requirement of containers are significantly heterogeneous.

A. Performance of CA-WFD

1) Algorithm Performance: We consider such a scenario where two new services (S_A and S_B) are being deployed into

TABLE IV
THE COSTS UNDER DIFFERENT PLACEMENT STRATEGIES

Data Centers	Algorithms	U_{cost}	B_{cost}	C_{cost}
DC_A	CA-WFD	0.069	0.703	0.403
	Random	0.114	0.824	0.400
	HA	0.111	0.709	0.424
	ENF	0.087	0.715	0.406
	Binpack	0.163	0.686	0.396
DC_B	CA-WFD	0.048	0.053	0.695
	Random	0.062	0.147	0.747
	HA	0.050	0.116	0.769
	ENF	0.042	0.090	0.758
	Binpack	0.088	0.170	0.698

DC_A and DC_B, respectively. S_A is instantiated as 2,000+ containers, and S_B is instantiated as 5,000+ containers. These containers have different resource requirements. In this set of experiments, to deploy these newly instantiated containers into data centers, CA-WFD is compared with four state-of-the-art container distribution strategies that are used by top container platform providers (e.g., Docker [33], Swarm [34] and Amazon [35]).

- **CA-WFD** is the Communication Aware Worst Fit Decreasing algorithm proposed in Section IV. In the evaluation, we set d as 2 in line 6 of Algorithm 1, i.e., two candidate servers are picked each time.
- **Random** assigns containers randomly. Random serves as a baseline in the evaluation.
- **HA** (High Availability) selects the server with the fewest containers of that service at the time of each container's deployment. HA is applied to optimize the load balance as well as service availability. However, it may also induce heavy communication overhead, because the containers spread over all the servers.
- **ENF** (Emptiest Node First) chooses the server with the fewest total containers. ENF aims at balancing the load of all the servers at a coarse granularity. Note that fewer containers do not definitely mean less resource utilization, since one big container (e.g., S_{B2} in Table III) can resume more resource than several small containers (e.g., S_{B1} in Table III).
- **Binpack** assigns containers to the server with the least available amount of CPU. Binpack tends to minimize the number of servers used.

Table IV compares the total cost after placing the containers with different algorithms. For clarity, in this section, the value of costs are after min-max normalization [36], and the lower and upper bounds are normalized to 0 and 1, respectively. The upper and lower bounds are calculated in the ideal conditions. Take Communication Cost for example, the upper bound of C_{cost} of a service is calculated when all the containers of this service spread on as many servers as possible, while the lower bound is calculated when these containers are all placed in the same server or the nearest servers (still under the Capacity Constraint, Conflict Constraint and Spread Constraint). With respect to Resource Utilization Cost (U_{cost}), CA-WFD performs overwhelmingly better than the other algorithms up to 57.7% in DC_A and up to 45.5% in DC_B. The second optimal algorithm is ENF, and the reason is that both CA-WFD and ENF tend to assign containers to servers with more free space, which benefits load balance. However, ENF regards the server with least containers as the "emptiest", which is imprecise. U_{cost} of Binpack is almost two times of CA-WFD, because Binpack assigns containers to least servers, which harms load balance. CA-WFD also achieves better or similar performance in both data centers in terms of Residual Resource Balance Cost (B_{cost}) and Communication Cost (C_{cost}), which confirms the effectiveness of CA-WFD. HA performs obviously worse than other designs in terms of C_{cost}, because HA spreads the containers among the servers which induces more cross-server communications.

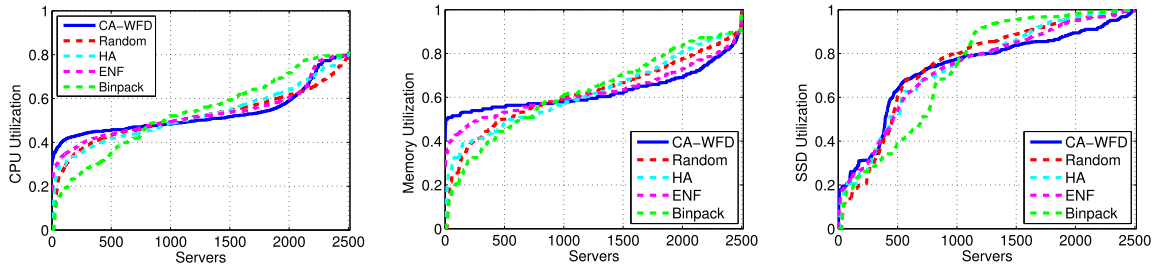
TABLE V
THE COSTS UNDER DIFFERENT ALGORITHM VARIATIONS

Data Centers	Designs	U_{cost}	B_{cost}	C_{cost}
DC_A	DR-WS	0.069	0.703	0.403
	WS-DP	0.072	0.705	0.405
	C-C	0.139	0.953	0.399
DC_B	DR-WS	0.048	0.053	0.695
	WS-DP	0.076	0.103	0.682
	C-C	0.091	0.244	0.671

Figure 4 shows CDF of resource utilization of DC_A servers. The horizontal axis represents the 2513 servers in DC_A and the vertical axis represents the utilization of three different resources. CA-WFD yields more balanced resource usages than other algorithms, which is consistent with the results in Table IV. As server resource utilization reflects the data center performance under stress tests, i.e., when burst occurs, we can expect better service throughput when placing containers by CA-WFD.

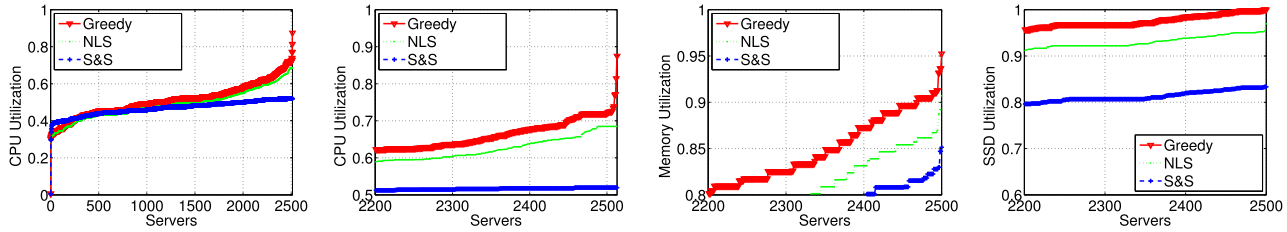
2) Algorithm Variations: As illustrated in Section IV, CA-WFD uses dominant requirement and weighted sum of residual resources to represent the "size" of containers and servers, respectively. Motivated by Panigrahy *et al.* [29], we evaluated several design choices in our experimental data centers, and the results perfectly confirmed the effectiveness of our design choice. In this section, we give the comparison between CA-WFD and another two representative variants, which run the same procedure as Algorithm 1 but with different metrics.

- **DR-WS** (Dominant Requirement – Weighted Sum), i.e., the design choice we adopt in Section IV.
- **WS-DP** (Weighted Sum – Dot Product) sorts the containers by the weighted sum of requirement vectors (i.e., $\sum_{r\in R} w_r \cdot D_c^r$), and selects the best server according to dot product of the vector of container requirement and the vector of residual resource of server (i.e., $\sum_{r\in R} a_r \cdot D_c^r \cdot A(h,r)$), where $a_r = \exp(0.01 \cdot avdem_r)$, and $avdem_r = \frac{1}{|R|}\sum_{r\in R} D_c^r$. Simulations in [29] show that WS-DP performs well in Vector Bin Packing [37].
- **C-C** (CPU – CPU) is a single-dimensional version Communication Aware Worst Fit Decreasing strategy, which only considers CPU utilizations when sorting and placing containers.



(a) CPU Utilization. (b) MEM Utilization. (c) SSD Utilization.

Fig. 4. The resource utilization of servers in DC_A under different placement strategies.



(a) CPU utilization. (b) CPU tail utilization. (c) Memory tail utilization. (d) SSD tail utilization.

Fig. 5. The resource utilizations of servers in DC_A under different reassignment strategies.

Table V compares the costs of CA-WFD and its variants. DR-WS (i.e., our design choice in Section IV-B) outperforms the other two designs in both data centers. A point worthing to note is that WS-DP performs obliviously worse in DC_B than in DC_A. We contribute this to WS-DP cannot effectively capture the resource features in more heterogeneous environments (note that in Table II, the server capacity of DC_A is more heterogeneous than that of DC_B). C-C assigns containers to the server with the maximum residual CPU, hence containers tend to be packed on servers with high end CPUs. This explains why it gains a slightly better result for C_{cost} than DR-WS and WS-DP, but poor performance for U_{cost} and B_{cost}. This implies that single-dimensional placement strategy is insufficient in real-world environments, because optimization of single resource easily brings poor utilization of other resources.

In summary, compared with the state-of-the-art algorithms, CA-WFD gains much balanced multi-resource utilization without inducing heavy communication overhead, which furtherly yields a better performance of services.

B. Performance of Sweep&Search

1) Algorithm Performance: We compared Sweep&Search with the following two alternative solutions, NLS and Greedy. Again, we evaluate these algorithms in experimental data centers for safety concerns.

- **Sweep&Search** (**S&S**) is the container reassignment algorithm we propose in Section V. To speed up the convergence of the $Search$ procedure in Algorithm 3, we empirically set w_u, w_b and w_c in Equation (6) as 1, $\frac{1}{|\mathbf{H}|}$, and $\frac{1}{|\mathbf{C}|^2}$, respectively, so that the three components of $Cost$ (i.e., $w_u * U_{cost}$, $w_b * B_{cost}$, and $w_c * C_{cost}$) fall in similar value ranges. Besides, we set δ in Sweep as 2%.

TABLE VI
THE COSTS UNDER DIFFERENT CONTAINER REASSIGNMENT ALGORITHMS

Data Centers	Algorithms	U_{cost}	B_{cost}	C_{cost}
DC_A	Sweep&Search	0.034	0.121	0.329
	NLS	0.049	0.356	0.351
	Greedy	0.057	0.390	0.362
DC_B	Sweep&Search	0.041	0.033	0.606
	NLS	0.052	0.168	0.630
	Greedy	0.062	0.121	0.647

- **NLS** is a noisy local search method, which is based on the winner team solution for Google Machine Reassignment Problem (GMRP) [16]. This method reallocates processes among a set of machines to improve the overall efficiency. In the evaluation, NLS adopts the same value of w_u, w_b and w_c as Sweep&Search in local searching.
- **Greedy** is a greedy algorithm, which tries to move containers from the "hottest" server to the "sparest" server each time. This algorithm reduces U_{cost} directly in a straightforward way.

Table VI shows the total costs produced by Sweep&Search, NLS and Greedy, separately. In DC_A, compared with Greedy (NLS), Sweep&Search achieves 40.4% (30.6%), 69.0% (66.0%) and 9.1% (6.2%) better performance in terms of U_{cost}, B_{cost} and C_{cost}, respectively. In DC_B, the benefits are 33.9%(21.2%), 72.7%(80.4%) and 6.3%(3.8%), respectively. The results show that Sweep&Search can jointly optimize communication overhead and balance resource utilizations.

Figure 5(a) shows the CDF of CPU utilization of the 2,513 servers in DC_A. The horizontal axis represents the 2513 servers in DC_A and the vertical axis represents the CPU utilization. There are about 330 servers whose CPU utilizations exceed 60% under the greedy algorithm and 210 servers under

TABLE VII
AVERAGE CPU UTILIZATION OF BOTTLENECK SERVERS

Algorithm	Top 1%	Top 5%	Top 10%
Sweep&Search	0.571	0.570	0.569
NLS	0.719	0.692	0.657
Greedy	0.736	0.703	0.675

TABLE VIII
THE COSTS UNDER DIFFERENT PARAMETER SETTINGS OF SWEEP&SEARCH IN DC_A

δ Value	U_{cost}	B_{cost}	C_{cost}
2%	0.034	0.121	0.329
10%	0.032	0.042	0.206
20%	0.033	0.034	0.168

the NLS algorithm. But when leveraging Sweep&Search, the highest CPU utilization falls down to 52%, which is much better than Greedy and NLS.

For high performance network services, the overall throughput of the system is generally determined by hot servers. We collect the resource usages of the top 300 hot servers produced by each algorithm, and the results are shown in Figure 5. Taking SSD as an example, the average utilization of the top 300 hot servers under greedy, NSL and Sweep&Search are 97.71%, 93.15% and 81.33%, respectively. To clearly show the quantified optimization results, the average CPU utilization of the top hot servers is shown in Table VII. The overall average CPU utilization of the 2,513 servers is 51.16%. We can see that Sweep&Search's performance is very close to the lower bound, and outperforms Greedy and NSL by up to 70%.

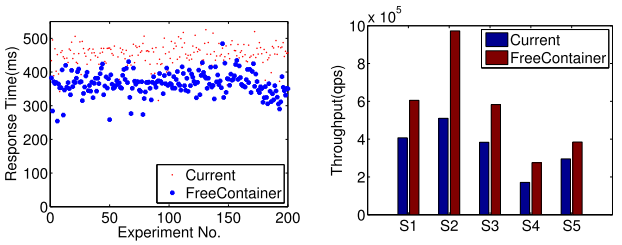
We attribute this to the following reasons. First, we take U_{cost} and B_{cost} into consideration to minimize the difference in resource utilizations and balance residual multiple resources. Second, the $Sweep$ stage makes room for the following search procedure, based on which the $Search$ stage could explore more branches to find better solutions.

2) Algorithm Efficiency: In this section, we show the effectiveness of *Sweep* and then evaluate the impacts of different parameter settings on the performance of Sweep&Search in DC_A. In the evaluation, we in turn set $\delta = 2\%(10\%, 20\%)$, which means that in each exploring iteration, we select $4\%(20\%, 40\%)$ servers as the set of candidates from the top $2\%(10\%, 20\%)$ and the tail $2\%(10\%, 20\%)$ and leverage neighbor searching on these candidate servers.

Table VIII shows the costs under different parameter settings. U_{cost}, B_{cost} and C_{cost} all benefit for a larger δ, which is consistent with the analysis in Appendix A. Especially, compared with 2%, by setting δ to 10% (20%), B_{cost} and C_{cost} are improved by 65.3% (71.9%) and 37.4% (48.9%), respectively. However, U_{cost} gains smaller improvement than B_{cost} and C_{cost}, which implies that a small δ can produce a good result in resource load balance. Note that although larger δ yields a reduction in the total cost, it also spends more time to sweep the servers in Algorithm 2.

VII. IMPLEMENTATION

In this section, we present the implementation of our solutions in Baidu. All the proposed algorithms have been implemented in a middleware product called FreeContainer [19], while FreeContainer is built in the data center orchestrator system that manages virtualized services.



(a) Request response time. (b) Throughput per service.

Fig. 6. System performance before and after deploying our solution.

TABLE IX
SERVICE INFORMATION

Service	Containers	Replicas per Container
S_1	3052 (large)	1 (small)
S_2	378 (medium)	6 (medium)
S_3	96 (small)	10 (large)
S_4	192 (medium)	3 (small)
S_5	98 (small)	6 (medium)

FreeContainer is deployed in an Internet data center with 6,000 servers, where 35 services are deployed together with some other background services. To evaluate the performance improvement of the data center after deploying our solution, we conduct a series of experiments to measure the following system features: response time, service throughput and resource utilization.

A. Response Time

The service response time refers to the total response time of a particular request. As each request would go through multiple containers, an efficient container communication scheme should give low network latency. We show the results in Figure 6(a), where the small red (big blue) points denote the request response time before (after) deploying our solution. The average response times are 451 ms and 365 ms, respectively. We can conclude that communication latency reduces up to 20% by container distribution optimization.

B. Service Throughput

To validate the online performance of the proposed algorithm, we perform stress test on this data center and measure the throughput of 5 representative services. The representative services are selected as follows. We classify the services into three types, i.e., small, medium and large, with respect to the number of containers and the mean number of replicas per container, respectively. Table IX summarizes the types of the services selected. Our intention is to cover as many service types as possible. The service throughput before and after deploying *FreeContainer* is shown in Figure 6(b). Taking S_2 as an example, the maximum throughput before deploying *FreeContainer* is 510,008 queries per second (qps), which raises to 972,581 qps after implementing our algorithms

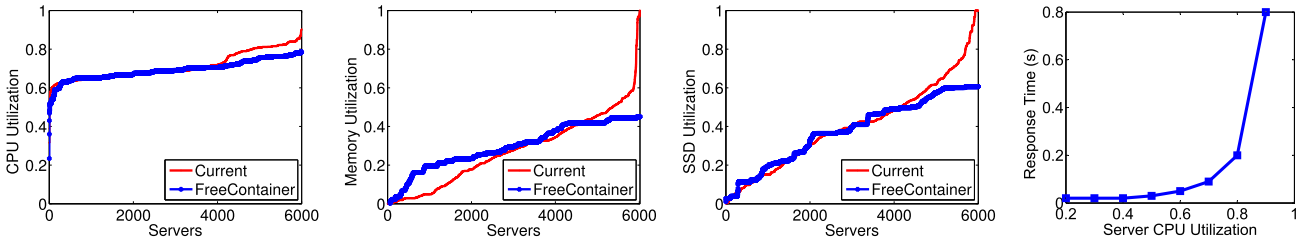


Fig. 7. (a) CDF of CPU utilizations of servers. (b) CDF of Memory utilization of servers. (c) CDF of SSD utilization of servers. (d) Impact of CPU utilization on response time.

(with an increase of 90%). We also observe that for S_5, the throughput is improved from 295,581 qps to 384,761 qps, with 30% increase. This is because there are only 588 interactive containers in S_5 but 2,268 containers in S_2. The results imply that the benefit is more significant for the service with more containers to communicate.

C. Resource Utilization

Resource utilization is another performance indicator. If resource utilization are balanced among servers, the throughput is generally also good. We measure resource utilizations under stress tests and show the results in Figure 7 (CPU in Figure 7(a), MEM in Figure 7(b) and SSD in Figure 7(c)). From these figures we can see that our solution eliminates the long tails of resource utilization. Taking CPU utilization as an example, there are about 800 servers whose utilization exceeds 80%. To show the influence caused by high resource utilization, we classify the servers according to CPU utilization (every 10 percent) and calculate the average response time of queries on these classified servers. The results show that when the average CPU utilization is below 60%, the latency keeps below $50ms$, but after that, the latency increases significantly with the increasing CPU utilization. For the servers with CPU utilization higher than 80%, the latency increases to $200ms$, and for the servers with CPU utilization higher than 90%, the latency increases to $800ms$ (16 times longer than that under 60% CPU utilization). Thus, the affected servers suffer from long request latency due to CPU resource shortage, and become the bottlenecks of the overall service performance. From the above results, we can conclude that our solution can lead to more balanced resource utilization.

VIII. Related Work

In this section, we survey some problems that are related to our problem, including Multi-Resource Generalized Assignment Problem, Google Machine Reassignment Problem, traffic-aware virtual machine placement Network Function Placement and Container Deployment and Migration.

A. Multi-Resource Generalized Assignment Problem (MRGAP)

MRGAP [25], [38] is an extension of the Generalized Assignment Problem (GAP) [39], [40], where multiple resources are associated with the items and bins. Solutions to MRGAP usually contains two phases. The first phase aims to obtain an initial feasible solution, and the second phase attempts to further improve the solution. Gavish and Pirkul [25] proposed two heuristics to generate the initial solution and a branch and bound algorithm to improve the solution. Privault and Herault [41] computed the initial solution by the bounded variable simplex method and optimized the solution by a simulated annealing algorithm. Mitrović-Minić and Punnen [30] and Yagiura *et al.* [42] generated a random initial solution in the first phase and adopted local searching techniques in the second phase. Mazzola and Wilcox [43] combined Pivot and Complement (P&C) and the heuristic proposed in [25] to obtain high-quality solutions. Shtub and Kogan [44] proposed a gradient descent based solution to the Dynamic MRGAP (DMRGAP), where the resource requirements of items change over time and an item can be assigned to several bins. Although we show that MRGAP is equivalent to the simplified CPP in Section IV-A, we emphasize that CPP and CRP are more complex than MRGAP because of the containerization-specific constraints (i.e., Conflict, Spread, Co-locate and Transient Constraints), which makes above solutions inapplicable in our scenarios.

B. Google Machine Reassignment Problem (GMRP)

GMRP was formulated by the Google research team as a subject of ROADEF/EURO Challenge, which aims to maximize the resource usage by reassigning processes among the the machines in data centers. Gavranović and Buljubašić [16] proposed the winner solution Noisy Local Search (NLS), which combines local searching techniques and noising strategy in reallocation. Different from NLS, we depart the reassignment into two steps, namely Sweep and Search. With the help of Sweep, we mitigate the hot hosts and obtain better initial conditions for the following local searching procedure. The evaluation result in Section VI shows that Sweep&Search yields significantly better results than directly applying local searching techniques.

C. Traffic-Aware Virtual Machine Placement

Like containerization, Virtual Machine (VM) is also a popular virtualization technique, where isolated operation systems run above a hypervisor layer on bare metals. Since each VM runs a full operating system [45], VMs usually

have bigger sizes and consumes more power than containers. Hence, traditional VM placement mainly concerns about optimization of energy consumption, resource utilizations and VM migration overhead [46]. Since the pioneer work of Meng *et al.* [47], many efforts have been made to mitigate inter-server communications by traffic-aware VM placement [48]–[57]. Meng *et al.* [47] defined the Traffic-aware VM Placement Problem and proposed a two-tier approximate algorithm to minimize inter-VM communications. Choreo [49] adopts a greedy heuristic to place VMs to minimize application completion time. Li *et al.* [50] proposed a series of traffic-aware VM placement algorithms to optimize traffic cost as well as single-dimensional resource utilization cost. Rui *et al.* [55] adopt a system optimization method to re-optimize VM distributions for joint optimization of resource load balancing and VM migration cost. Different from these work, we optimize both communication overhead and multi-resource load balancing. Besides, since containers can be deployed in VMs instead of physical machines, the solutions proposed in our paper are orthogonal to these VM placement strategies. Therefore, VM resource utilization and inter-VM communications could be optimized by container placement/reassignment, and that of physical machines could be optimized by VM placement.

D. Network Function Placement

Network Function Virtualization (NFV) has recently gained wide attention from both industry and academia, making the study of their placement a popular research topic [17], [58]–[71]. Wang *et al.* [17] studied the flow-level multi-resource load balancing problem in NFV and proposed a distributed solution based on the proximal Jacobian ADMM (Alternating Direction Method of Multipliers). Marotta and Kassler [61] proposed a mathematical model based on the Robust Optimization theory to minimize the power consumption of the NFV infrastructure. In [65], an affinity-based heuristic is proposed to minimize inter-cloud traffic and response time. Zhang *et al.* [66] proposed a Best Fit Decreasing based heuristic algorithm to place network functions to achieve high utilization of single dimensional sources. Taleb *et al.* [67] studied the network function placement problem from many aspects, including minimizing path between users and their respective data anchor gateways, measuring existing NFV placement algorithms [67], placing Packet Data Network (PDN) Gateway network functionality and Evolved Packet Core (EPC) in the cloud [68], [69], [71], and modeling cross-domain network slices for 5G [70]. Since network functions work in chains and containers are deployed by groups, the communication patterns are totally different between the two systems. Hence, the communication optimization solutions in NFV is non-applicable to container placement. Besides, none of these work aims at the joint optimization of communication overhead and multi-resource load balancing in data centers.

E. Container Deployment and Migration

A lot of work has been studied to deploy containers among virtual machines or physical machines for various optimization purposes. Zhang *et al.* [72] proposed a novel container placement strategy for improving physical resource utilization. The works [73]–[75] studied the container placement problem for minimizing energy consumption in the cloud. Mao *et al.* [76] presented a resource-aware placement scheme to improve the system performance in a heterogeneous cluster. Nardelli *et al.* [77] studied the container placement problem for optimizing deployment cost. However, none of the above work considers the communication cost among containers.

The container migration issues also have been extensively studied in the literature. The first part of the related works concentrate on developing container live migration techniques. The works [78], [79] proposed solutions for live migrating Linux containers, while Pickartz *et al.* [80] proposed the techniques for live migrating Docker containers. The prior works [81]–[83] further optimized the existing container migration techniques for reducing migration overhead. The second part of the related works focused on the container migration strategies. Li *et al.* [84] aimed to achieve load balancing of cloud resources through container migration. Guo and Yao [85] proposed a container scheduling strategy based on neighborhood division in micro service, with the purpose of reducing the system load imbalance and improve the overall system performance. Kaewkasi and Chuenmuneewong [86] applied the ant cology optimization (ACO) in the context of container scheduling, which aimed to balance the resource usages and achieve better performance. Xu *et al.* [87] proposed a resource scheduling approach for the container virtualized cloud environments to reduce response time of customers jobs and improve resource utilization. Again, none of the above work considers communication cost among containers.

IX. Conclusion

More and more Internet service providers deploy their services in containers due to the promising properties of containerization. However, applying containerization in large-scale Internet data centers faces the trade-off between communication cost and multi-resource load balance.

In this paper, we go into the container distribution problem in large-scale data centers and break it down into two stages, i.e., Container Placement Problem and Container Reassignment Problem, which are both NP-hard. For Container Placement, we propose an efficient heuristic named Communication Aware Worst Fit Decreasing which extends WFD to CPP by considering both multiple resource load balance as well as communication overhead reduction. For Container Reassignment Problem, we design a two-stage algorithm called Sweep&Search to re-optimize the container distribution, which firstly handles overloaded servers and then optimizes the objectives by local search techniques.

Extensive experiments have been conducted to evaluate our algorithms. The results show that our algorithms outperform the state-of-the-art solutions up to 70%. We further implemented our solutions in a data center with more than 6,000 servers and 35 services, and the measurements indicate that our solutions can effectively reduce the communication overhead among interactive containers while simultaneously increasing the overall service throughput up to 90%.

APPENDIX A
APPROXIMATION ANALYSIS OF SWEEP&SEARCH

In this section, we would like to prove that the output of Sweep&Search is $(1+\epsilon,\theta)$-approximate to the theoretical optimum result P^* (for simplification, we use $\hat{P}$ instead of P_{best} in the subsequent analysis), where ϵ is an accuracy parameter, and θ is a confidence parameter that represents the possibility of that accuracy [88]. More specifically, this $(1+\epsilon,\theta)$-approximation can be formulated as the following inequality:

$$Pr[|\hat{P}-P^*|\leq \epsilon P^*]\geq 1-\theta. \tag{14}$$

If $\epsilon = 0.05$ and $\theta = 0.1$, it means that the output of Sweep&Search $\hat{P}$ differs from the optimal solution P^* by at most 5% (the accuracy bound) with a probability 90% (the confidence bound).

Sweep introduces no deviation, so the deviation of $\hat{P}$ and P^* mainly comes from the *Search* stage which consists of two parts: one is to select a subset of host set $\mathbf{H}$ to run *Search*, the other is the stopping condition. Specifically, the core idea of the *search* algorithm is to select some candidate hosts from $\mathbf{H}$, expand to search three kinds of neighbors for a few iterations and generate an approximate result in each iteration.

Let b_i be the branch that is explored on h_i and x_i be the minimum cost of b_i. Assume there are n branches in total, a fact that can be easily seen is that the optimal result (minimum cost) is $Q^* = min\{x_1,...,x_n\}$. Given an approximation ratio ϵ, we would like to prove that the output of Sweep&Search $\hat{P}$ with cost $\hat{Q}$ meets $|\hat{Q}-Q^*|\leq \epsilon Q^*$ with a bounded probability. We split the total error ϵ into two parts and try to bound the above two errors separately.

A. Bound the Error From Stopping Conditions

In each iteration, we explore 2δ branches. Let $\hat{Q}_{iter} = min\{min\{x_1,...,x_{2\delta}\}, \frac{(1-\epsilon)Q^*_{iter}}{2}\}$, which means that the stopping condition is a balance of the following two: 1) the minimum cost on these branches reaches the best result; and 2) the threshold $\frac{1-\epsilon}{2}$ is reached.

Lemma 1: The output $\hat{Q}_{iter}$ in each iteration satisfies $|\hat{Q}_{iter}-Q^*_{iter}|\leq \frac{\epsilon}{2}Q^*_{iter}$

Proof: There are two parts: 1) If the minimum cost on these 2δ branches reaches the current best one, then $\hat{Q}_{iter} = Q^*_{iter}$; 2) If the minimum cost on these branches is greater than the current best, then $\hat{Q}_{iter} = \frac{1-\epsilon}{2}Q^*_{iter}$.

Overall, $|\hat{Q}_{iter}-Q^*_{iter}|\leq |\frac{1-\epsilon}{2}Q^*_{iter}-Q^*_{iter}| = \frac{\epsilon}{2}Q^*_{iter}$, so the deviation is bounded by $\frac{\epsilon}{2}$.

B. Bound the Error From Subset Selection

We now try to prove that we can bound $|\hat{Q}-Q^*|$ by exploring 2δ hosts in each iteration.

Before giving the proof, we first introduce the Hoeffding Bound:

Hoeffding Inequality: There are k random identical and independent variables V_i. For any ε, we have

$$Pr[|V-E(V)|\geq \varepsilon]\leq e^{-2\varepsilon^2 k}. \tag{15}$$

With this Hoeffding Bound, we have the following lemma:

Lemma 2: There is an upper bound for $Pr[|\hat{Q}-Q^*|\leq \frac{\epsilon}{2}Q^*]$ when exploring 2δ hosts in each iteration.

Proof: Assume the minimum cost of all branches is in uniform distribution (range from a to b), so we have $\hat{Q} = (min\{x_1,...,x_{2\delta}\})$ and $E(\hat{Q}) = (1-\frac{x_i}{b-a})^{2\delta}$. Let $Y_i = 1-\frac{x_i}{b-a}$, and $Y = \prod_{i=1}^{2\delta} Y_i$, we have

$$E(\hat{Q}) = Y < Y^{\frac{n}{2\delta}}. \tag{16}$$

As Y is associated with $\hat{Q}$ and the expectation of the minimum Y_i is associated with Q^*, so $\hat{Q}$ and Q^* are linked together. More specifically, $E(Y) = E(Y_i)^{2\delta}$, $E(Y_i) = E(Y)^{\frac{1}{2\delta}}$. Thus, we have

$$E(Q^*) = (1-\frac{x_i}{b-a})^n = E(Y_i)^n = E(Y)^{\frac{n}{2\delta}} \tag{17}$$

and

$$Pr[|E(\hat{Q})-E(Q^*)|\geq \frac{\epsilon}{2}] < Pr[|Y^{\frac{n}{2\delta}}-E(Y)^{\frac{n}{2\delta}}|\geq \frac{\epsilon}{2}]. \tag{18}$$

Through the above Hoeffding Bound (Inequation 15), we have

$$Pr[|Y^{\frac{n}{2\delta}}-E(Y)^{\frac{n}{2\delta}}|\geq \frac{\epsilon}{2}]\leq e^{-2(\frac{\epsilon}{2})^2*2\delta}. \tag{19}$$

Finally,

$$Pr[|E(\hat{Q})-E(Q^*)|\geq \frac{\epsilon}{2}]\leq e^{-\epsilon^2\delta}. \tag{20}$$

Now we can combine the above two errors: the error rate from stopping condition is within $\frac{\epsilon}{2}$ and the error rate from subset selection is also bounded to $\frac{\epsilon}{2}$ within a probability. So we can propose the whole theorem as follows.

Theorem: Let Q^* be the theoretical optimal result (with the minimum overall cost) of the container group reassignment problem, Sweep&Search can output an approximate result $\hat{Q}$ where $|\hat{Q}-Q^*|\leq \epsilon Q^*$ with probability at least $e^{-\epsilon^2\delta}$.

As a conclusion, we can give an upper bound to the deviation and the possibility is associated with the number of selected hosts in each searching iteration. With a given accuracy, we can further improve that probability by exploring more hosts.

ACKNOWLEDGMENTS

This work was partly done during Liang Lv and Yuchao Zhang's internship in Baidu.

REFERENCES

[1] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proc. ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 275–287.

[2] (2016). *Docker*. [Online]. Available: http://www.docker.com/

[3] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar, "FreeFlow: High performance container networking," in *Proc. ACM 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 43–49.

[4] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *Proc. 8th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2016, pp. 1–6.

[5] Y. Zhang, K. Xu, H. Wang, Q. Li, T. Li, and X. Cao, "Going fast and fair: Latency optimization for cloud-based service chains," *IEEE Netw.*, vol. 32, no. 2, pp. 138–143, Mar./Apr. 2018.

[6] M. Shen, B. Ma, L. Zhu, R. Mijumbi, X. Du, and J. Hu, "Cloud-based approximate constrained shortest distance queries over encrypted graphs with privacy protection," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 4, pp. 940–953, Apr. 2018.

[7] Y. Zhang, K. Xu, H. Wang, and M. Shen, "Towards shorter task completion time in datacenter networks," in *Proc. IEEE 34th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2015, pp. 1–8.

[8] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using Mantri," in *Proc. OSDI*, vol. 10, no. 1, 2010, pp. 265–278.

[9] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. 11th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2014, pp. 583–598.

[10] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 1, pp. 97–107, Jan. 2014.

[11] Y. Zhang, K. Xu, G. Yao, M. Zhang, and X. Nie, "PieBridge: A cross-DR scale large data transmission scheduling system," in *Proc. Conf. ACM SIGCOMM Conf.*, 2016, pp. 553–554.

[12] Y. Zhang *et al.*, "BDS: A centralized near-optimal overlay network for inter-datacenter data replication," in *Proc. ACM 13th EuroSys Conf.*, 2018, Art. no. 10.

[13] K. Xu *et al.*, "Modeling, analysis, and implementation of universal acceleration platform across online video sharing sites," *IEEE Trans. Services Comput.*, vol. 11, no. 3, pp. 534–548, May/Jun. 2018.

[14] H. Wang *et al.*, "Toward cloud-based distributed interactive applications: Measurement, modeling, and analysis," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 3–16, Feb. 2018.

[15] Y. Zhang, K. Xu, X. Shi, H. Wang, J. Liu, and Y. Wang, "Design, modeling, and analysis of online combinatorial double auction for mobile cloud computing markets," *Int. J. Commun. Syst.*, vol. 31, no. 7, p. e3460, 2018.

[16] H. Gavranović and M. Buljubašić, "An efficient local search with noising strategy for Google machine reassignment problem," *Ann. Oper. Res.*, vol. 242, pp. 19–31, Jul. 2014.

[17] T. Wang, H. Xu, and F. Liu, "Multi-resource load balancing for virtual network functions," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, Jun. 2017, pp. 1322–1332.

[18] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proc. ACM 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 13.

[19] Y. Zhang *et al.*, "A communication-aware container re-distribution approach for high performance VNFs," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, Jun. 2017, pp. 1555–1564.

[20] (2016). *Freebds.chrootcfreebdsmanpages*. [Online]. Available: http://www.freebsd.org/cgi/man.cgi

[21] W. Felter, A. P. Ferreira, R. Rajamony, and J. C. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2015, pp. 171–172.

[22] (2016). *Kubernetes*. [Online]. Available: http://kubernetes.io/

[23] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. NSDI*, 2011, pp. 295–308.

[24] R. T. Marler and J. S. Arora, "Survey of multi-objective optimization methods for engineering," *Struct. Multidisciplinary Optim.*, vol. 26, no. 6, pp. 369–395, Apr. 2004.

[25] B. Gavish and H. Pirkul, "Algorithms for the multi-resource generalized assignment problem," *Manage. Sci.*, vol. 37, no. 6, pp. 695–713, 1991.

[26] S. Sahni and T. Gonzalez, "P-complete approximation problems," *J. ACM*, vol. 23, no. 3, pp. 555–565, 1976.

[27] D. S. Johnson, "Fast algorithms for bin packing," *J. Comput. Syst. Sci.*, vol. 8, no. 3, pp. 272–314, 1974.

[28] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno, "Resource allocation in distributed mixed-criticality cyber-physical systems," in *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst.*, Jun. 2010, pp. 169–178.

[29] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," Microsoft Res., Silicon Valley, CA, USA, Tech. Rep., Jan. 2011.

[30] S. Mitrović-Minić and A. P. Punnen, "Local search intensified: Very large-scale variable neighborhood search for the multi-resource generalized assignment problem," *Discrete Optim.*, vol. 6, no. 4, pp. 370–377, 2009.

[31] J. A. Dıaz and E. Fernández, "A Tabu search heuristic for the generalized assignment problem," *Eur. J. Oper. Res.*, vol. 132, no. 1, pp. 22–38, 2001.

[32] R. Masson *et al.*, "An iterated local search heuristic for multi-capacity bin packing and machine reassignment problems," *Expert Syst. Appl.*, vol. 40, no. 13, pp. 5266–5275, 2013.

[33] (2017). *Container Distribution Strategies*. [Online]. Available: https://docs.docker.com/docker-cloud/infrastructure/deployment-strategies/

[34] (2017). *Docker Swarm Strategies*. [Online]. Available: https://docs.docker.com/swarm/scheduler/strategy/

[35] AEC Service. (2017). *Amazon ECS Task Placement Strategies*. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-placement-strategies.html

[36] J. Han, *Data Mining: Concepts and Techniques*. San Mateo, CA, USA: Morgan Kaufmann, 2005.

[37] H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali, "Approximation and online algorithms for multidimensional bin packing: A survey," *Comput. Sci. Rev.*, vol. 24, pp. 63–79, May 2017.

[38] H. Pirkul, "Computer and database location in distributed computer systems," Ph.D. dissertation, Eastman School Music, Univ. Rochester, Rochester, NY, USA, 1983.

[39] G. T. Ross and R. M. Soland, "A branch and bound algorithm for the generalized assignment problem," *Math. Program.*, vol. 8, no. 1, pp. 91–103, Dec. 1975.

[40] T. Öncan, "A survey of the generalized assignment problem and its applications," *Inf. Syst. Oper. Res.*, vol. 45, no. 3, pp. 123–141, 2007.

[41] C. Privault and L. Herault, "Solving a real world assignment problem with a metaheuristic," *J. Heuristics*, vol. 4, no. 4, pp. 383–398, Dec. 1998.

[42] M. Yagiura, S. Iwasaki, T. Ibaraki, and F. Glover, "A very large-scale neighborhood search algorithm for the multi-resource generalized assignment problem," *Discrete Optim.*, vol. 1, no. 1, pp. 87–98, 2004.

[43] J. B. Mazzola and S. P. Wilcox, "Heuristics for the multi-resource generalized assignment problem," *Nav. Res. Logistics*, vol. 48, no. 6, pp. 468–483, 2001.

[44] A. Shtub and K. Kogan, "Capacity planning by the dynamic multi-resource generalized assignment problem (DMRGAP)," *Eur. J. Oper. Res.*, vol. 105, no. 1, pp. 91–99, 1998.

[45] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proc. Int. MIDDLEWARE Conf.*, 2016, Art. no. 1.

[46] Z. Á. Mann and M. Szabó, "Which is the best algorithm for virtual machine placement optimization?" *Concurrency Comput. Pract. Exper.*, vol. 29, no. 10, p. e4083, 2017.

[47] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–9.

[48] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the network in cloud computing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 187–198, 2012.

[49] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware task placement for cloud applications," in *Proc. Conf. Internet Meas. Conf.*, 2013, pp. 191–204.

[50] X. Li, J. Wu, S. Tang, and S. Lu, "Let's stay together: Towards traffic aware virtual machine placement in data centers," in *Proc. IEEE INFOCOM*, Apr./May 2014, pp. 1842–1850.

[51] T. Ma, J. Wu, Y. Hu, and W. Huang, "Optimal VM placement for traffic scalability using Markov chain in cloud data centre networks," *Electron. Lett.*, vol. 53, no. 9, pp. 602–604, 2017.

[52] Y. Zhao, Y. Huang, K. Chen, M. Yu, S. Wang, and D. Li, "Joint VM placement and topology optimization for traffic scalability in dynamic datacenter networks," *Comput. Netw.*, vol. 80, pp. 109–123, Apr. 2015.

[53] A. Rai, R. Bhagwan, and S. Guha, "Generalized resource allocation for the cloud," in *Proc. ACM Symp. Cloud Comput.*, 2012, Art. no. 15.

[54] L. Wang *et al.*, "GreenDCN: A general framework for achieving energy efficiency in data center networks," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 1, pp. 4–15, Jan. 2014.

[55] R. Li, Q. Zheng, X. Li, and J. Wu, "A novel multi-objective optimization scheme for rebalancing virtual machine placement," in *Proc. IEEE Int. Conf. Cloud Comput.*, Jun./Jul. 2016, pp. 710–717.

[56] L. Gu, D. Zeng, S. Guo, Y. Xiang, and J. Hu, "A general communication cost optimization framework for big data stream processing in geo-distributed data centers," *IEEE Trans. Comput.*, vol. 65, no. 1, pp. 19–29, 2015.

[57] M. Shen, K. Xu, F. Li, K. Yang, L. Zhu, and L. Guan, "Elastic and efficient virtual network provisioning for cloud-based multi-tier applications," in *Proc. IEEE 44th Int. Conf. Parallel Process. (ICPP)*, Sep. 2015, pp. 929–938.

[58] T. Taleb, M. Bagaa, and A. Ksentini, "User mobility-aware virtual network function placement for virtual 5G network infrastructure," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2015, pp. 3879–3884.

[59] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and placing chains of virtual network functions," in *Proc. IEEE Int. Conf. Cloud Netw.*, Oct. 2014, pp. 7–13.
[60] K. Kawashima, T. Otoshi, Y. Ohsita, and M. Murata, "Dynamic placement of virtual network functions based on model predictive control," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Apr. 2016, pp. 1037–1042.
[61] A. Marotta and A. Kassler, "A power efficient and robust virtual network functions placement problem," in *Proc. Teletraffic Congr.*, 2017, pp. 331–339.
[62] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *Proc. IEEE Int. Conf. Cloud Netw.*, Oct. 2015, pp. 171–177.
[63] F. Wang, R. Ling, J. Zhu, and D. Li, "Bandwidth guaranteed virtual network function placement and scaling in datacenter networks," in *Proc. IEEE Int. Perform. Comput. Commun. Conf.*, Dec. 2015, pp. 1–8.
[64] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *Proc. IEEE Int. Conf. Cloud Netw.*, Oct. 2015, pp. 255–260.
[65] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Optimal virtual network function placement in multi-cloud service function chaining architecture," *Comput. Commun.*, vol. 102, pp. 1–16, Apr. 2017.
[66] Q. Zhang, Y. Xiao, F. Liu, J. C. S. Lui, J. Guo, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, Jun. 2017, pp. 731–741.
[67] A. Laghrissi, T. Taleb, M. Bagaa, and H. Flinck, "Towards edge slicing: VNF placement algorithms for a dynamic & realistic edge cloud environment," in *Proc. IEEE Global Commun. Conf.*, Dec. 2017, pp. 1–6.
[68] J. Prados-Garzon, A. Laghrissi, M. Bagaa, and T. Taleb, "A queuing based dynamic auto scaling algorithm for the LTE EPC control plane," in *Proc. IEEE Global Commun. Conf.*, Dec. 2018, pp. 1–6.
[69] M. Bagaa, T. Taleb, and A. Ksentini, "Service-aware network function placement for efficient traffic handling in carrier cloud," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2014, pp. 2402–2407.
[70] M. B. D. D. R. A. Addad, T. Taleb, and H. Flinck, "Towards modeling cross-domain network slices for 5G," in *Proc. IEEE Global Commun. Conf.*, Dec. 2018, pp. 1–6.
[71] M. Bagaa, T. Taleb, A. Laghrissi, and A. Ksentini, "Efficient virtual evolved packet core deployment across multiple cloud domains," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2018, pp. 1–6.
[72] R. Zhang, A.-M. Zhong, B. Dong, F. Tian, and R. Li, "Container-VM-PM architecture: A novel architecture for docker container placement," in *Proc. Int. Conf. Cloud Comput.* Seattle, WA, USA: Springer, Jun. 2018, pp. 128–140.
[73] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A framework and algorithm for energy efficient container consolidation in cloud data centers," in *Proc. IEEE Int. Conf. Data Sci. Data Intensive Syst. (DSDIS)*, Dec. 2015, pp. 368–375.
[74] Z. Dong, W. Zhuang, and R. Rojas-Cessa, "Energy-aware scheduling schemes for cloud data centers on Google trace data," in *Proc. IEEE Online Conf. Green Commun. (OnlineGreenComm)*, Nov. 2014, pp. 1–6.
[75] T. Shi, H. Ma, and G. Chen, "Energy-aware container consolidation based on PSO in cloud data centers," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2018, pp. 1–8.
[76] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, "DRAPS: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster," in *Proc. IEEE 36th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2017, pp. 1–8.
[77] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic provisioning of virtual machines for container deployment," in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. Companion*, 2017, pp. 5–10.
[78] Y. Qiu, "Evaluating and improving LXC container migration between cloudlets using multipath TCP," Ph.D. dissertation, Dept. Elect. Comput. Eng., Carleton Univ., Ottawa, ON, Canada, 2016.
[79] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Live service migration in mobile edge clouds," *IEEE Wireless Commun.*, vol. 25, no. 1, pp. 140–147, Feb. 2018.
[80] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti, "Migrating LinuX containers using CRIU," in *Proc. Int. Conf. High Perform. Comput.* Frankfurt, Germany: Springer, Jun. 2016, pp. 674–684.
[81] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Trans. Mobile Comput.*, to be published, doi: 10.1109/TMC.2018.2871842.
[82] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, Oct. 2017, Art. no. 11.
[83] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 2137–2142.
[84] P. Li, H. Nie, H. Xu, and L. Dong, "A minimum-aware container live migration algorithm in the cloud environment," *Int. J. Bus. Data Commun. Netw.*, vol. 13, no. 2, pp. 15–27, 2017.
[85] Y. Guo and W. Yao, "A container scheduling strategy based on neighborhood division in micro service," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Apr. 2018, pp. 1–6.
[86] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *Proc. IEEE 9th Int. Conf. Knowl. Smart Technol. (KST)*, Feb. 2017, pp. 254–259.
[87] 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. (CSE)*, Dec. 2014, pp. 257–264.
[88] Z. Han, M. Hong, and D. Wang, *Signal Processing and Networking for Big Data Applications*. Cambridge, U.K.: Cambridge Univ. Press, 2017.

Liang Lv received the B.S. and M.S. degrees from the School of Computer, National University of Defense Technology, Hunan, China, in 2008 and 2010, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Tsinghua University. His research interests include network virtualization and resource management.

Yuchao Zhang received the B.S. degree in computer science and technology from Jilin University in 2012 and the Ph.D. degree from the Computer Science Department, Tsinghua University, in 2017. She is currently with the Beijing University of Posts and Telecommunications. Her research interests include large-scale datacenter network, content delivery network, data-driven network, and edge computing.

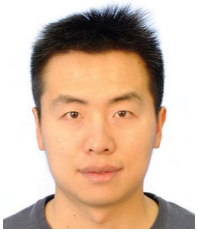
Yusen Li received the Ph.D. degree in computer science from Nanyang Technological University in 2013. He is currently an Associate Professor with the College of Computer Science, Nankai University, China. His research interests include resource allocation and scheduling issues in distributed systems and cloud computing.

Ke Xu (M'02–SM'09) received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University. He is currently a Full Professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include next-generation Internet, P2P systems, Internet of Things, network virtualization, and network economics. He is a member of ACM. He serves as an Associate Editor for the IEEE INTERNET OF THINGS JOURNAL. He has guest edited several special issues in IEEE and springer journals.

Dan Wang (S'05–M'07–SM'13) received the B.Sc. degree in computer science from Peking University, Beijing, China, the M.Sc. degree in computer science from Case Western Reserve University, Cleveland, OH, USA, and the Ph.D. degree in computer science from Simon Fraser University, Vancouver, BC, Canada. He is currently an Associate Professor with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. His current research interests include Internet architecture and QoS, smart buildings, and green computing.

Wendong Wang (M'05) received the B.E. and M.E. degrees from the Beijing University of Posts and Telecommunications, China, in 1985 and 1991, respectively. He is currently a Full Professor with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. He has published over 200 papers in various journals and conference proceedings. His current research interests are the next-generation network architecture, network resources management and QoS, and mobile Internet. He is a member of the IEEE.

Minghui Li received the M.S. degree from Zhejiang University, Zhejiang, China. He is currently a Site Reliability Engineer with Baidu. His research interests include large-scale distributed system and site reliability engineering.



Xuan Cao received the M.S. degree from the Nanjing University of Science and Technology, Nanjing, China. He is currently a Software Engineer with Baidu. His research interests include large-scale distributed system and site reliability engineering.

Qingqing Liang received the M.S. degree from the Nanjing University of Posts and Telecommunications, Nanjing, China. His research interests include large-scale distributed system and site reliability engineering.