A Communication-aware Container Re-distribution Approach for High Performance VNFs

Yuchao Zhang[1,2], Yusen Li[3,4], Ke Xu[1,5], Dan Wang[2], Minghui Li[4], Xuan Cao[4], Qingqing Liang[4]
[1]Tsinghua University [2]Hong Kong Polytechnic University [3]Nankai University
[4]Baidu [5]Tsinghua National Laboratory for Information Science and Technology
zhangyc14@mails.tsinghua.edu.cn, liyusen@nbjl.nankai.edu.cn, xuke@tsinghua.edu.cn,
dan.wang@polyu.edu.hk, {liminghui,caoxuan,liangqingqing}@baidu.com

***Abstract*—Containers have been used in many applications for isolation purposes due to the lightweight, scalable and highly portable properties. However, to apply containers in virtual network functions (VNFs) faces a big challenge because high-performance VNFs often generate frequent communication workloads among containers while the container communications are generally not efficient. Compared with hardware modification solutions, properly distributing containers among hosts is an efficient and low-cost way to reduce communication overhead. However, we observe that this approach yields a trade-off between the communication overhead and the overall throughput of the cluster. In this paper, we focus on the communication-aware container re-distribution problem to optimize the communication overhead and the overall throughput jointly for VNF clusters. We propose a solution called *FreeContainer* which utilizes a novel two-stage algorithm to re-distribute containers among hosts. We implement *FreeContainer* in Baidu clusters with 6000 servers and 35 services deployed. Extensive experiments on real networks are conducted to evaluate the performance of the proposed approach. The results show that *FreeContainer* can increase the overall throughput up to 90% with significant reduction on communication overhead.**

***Index Terms*—Container Communication, High-performance VNF, System Throughput**

I. Introduction

Containerization has become a popular virtualization technology since containers are lightweight, scalable and highly portable, offering good isolation using control group (cgroup) [1] and namespace [2]. Many applications thus are being developed, deployed and managed as containers. Some promising projects such as OpenStack [3], OpenVZ [4], FreeBSD Jails [5] and Solaris Zones [6] have already published their container-supporting versions.

However, containers are now faced with great challenges when being adopted to build virtualized network functions (VNFs), especially for high-performance VNFs managed on the hypervisor-based platforms (such as Xen [7], KVM [8], Hyper-V [9] and VMWare Server [10]). This is because VNFs are usually built as groups of containers that communicate with each other to deliver the desired service, which requires highly efficient communications among containers. But the host networks introduce extra communication overhead to the VNFs [11], and the containers deployed on different hosts usually communicate in extremely low efficiency, which significantly degrades the VNF performance especially in large scale systems [12].

To reduce the communication overhead, some kernel bypassing techniques like RDMA [13], DPDK [14] and FreeFlow [11] are proposed. While these techniques can improve the communication efficiency, they sacrifice some isolations, which is the key advantage of containerization. To retain isolations, a commonly used way for communication overhead reduction is to find a proper container distribution among hosts. If the containers in the same group are deployed on the same host or hosts nearby, the communication overhead will be greatly reduced. But this approach yields a trade-off between the communication overhead and the overall throughput.

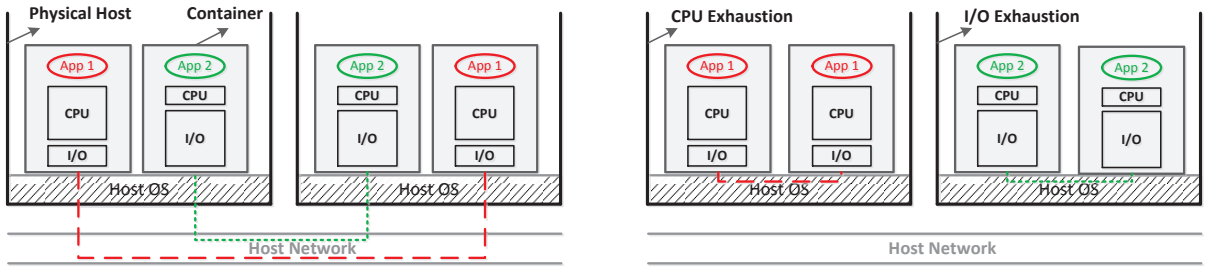
As we know, containers of the same service are generally intensive to the same resource (e.g., containers for a big data analytics application are all CPU intensive [15]–[17]). Assigning the containers of the same service on the same host may cause heavily imbalanced resource utilizations on hosts, which will significantly degrade the overall throughput.

TABLE I: CPU Utilization in a Cluster with 3000 Servers.

Top 1%	Top 5%	Top 10%	Mean
0.837	0.704	0.675	0.52

Example. We bring a real example to show the conflicts between the communication overhead and the resource utilizations. We conduct measurements on a Baidu [18] cluster with over 3000 servers. In this cluster, 25 applications with different resource intensions are mixed-deployed. The containers of the same application (i.e., belong to the same group) are deployed closer. Table I shows the top 1%, 5%, 10% servers in terms of CPU utilizations under stress test. It is clear that the CPU utilizations are highly imbalanced since the top 1% servers have CPU utilizations over 80% while the mean CPU utilization of all the servers is only 52%.

The above example is not a special case because the conflict between container communication and host resource utilization is ubiquitous in most server providers. Figure 1 gives a more clear illustration of the reason that causes the conflicts. Two types of applications are deployed on two hosts, and each application has two containers. App 1 is CPU-intensive (maybe a big data analytics application [16], [17]) and App 2 is



(a) High network communication overhead, long latency.

(b) Low host resource utilizations, low throughput.

Fig. 1: The conflict between container communication and host resource utilization.

network I/O-intensive (maybe a data transfer application [19], [20]). Figure 1(a) shows the mix-deployment solution which assigns the containers of different applications on the same host. This approach achieves high resource utilizations for both CPU and network, while incurs high communication overhead between two hosts. Figure 1(b) shows another solution which assigns the containers of the same application on the same host. The communication overhead is significantly reduced, however, the utilizations of CPU and network are highly imbalanced at the two hosts.

In this paper, we address the container distribution issues in VNFs, which aims to optimize both communication overhead and overall throughput. We consider a general scenario, where an initial container distribution is given, and our objective is to re-distribute the containers among hosts by container migrations. Re-distribution problem is more challenging. This is because online services cannot be interrupted during container migrations, which requires some resources (called transient resources) are consumed both at the original host and the new host. For this reason, some containers (especially for large containers) may not be able to migrate due to resource limitations. To address this issue, we propose a communication-aware container re-distribution solution called *FreeContainer*, which separates the entire re-distribution into two stages. In the first stage, we try to handle the "hot hosts" (i.e., the hosts with high resource utilizations), which are generally the bottleneck of the overall throughput. After that, local search is used to further improve the solution in the second stage. *FreeContainer* does not require hardware modification and is completely transparent to online applications. We implement *FreeContainer* on Baidu's clusters and conduct extensive evaluations in real environment. The results show that *FreeContainer* can significantly reduce the communication overhead and improve the overall throughput compared to the existing approaches.

The rest of this paper is structured as follows. Section II covers related work. Section III describes the background and formulates the container re-instantiation problem. We introduce the *FreeContainer* design in detail in Section IV and give the algorithm analysis in Section V. The implementation and evaluation are provided in Section VI and VII, respectively. At last, Section VIII concludes the paper.

II. Related Work

As the popularity of NFV and container grows, a plenty of related work has been conducted, which includes NVF platforms, container communication and VM technologies.

NFV Platforms. There are two options to isolate VNFs running on the same host: hypervisors and containers [21]. Hypervisors support different guest OSes on the same hardware, but the performance and portability is not good. Container-based platforms offer better performance and portability by using namespace and cgroup technologies. Some projects such as OpenStack [3] have already published the NFV-supporting versions, and they are typically managed by a central orchestrator (e.g., Kubernetes [22], Matrix [23]). These orchestrators provide us a roadmap of deployment-specific information of containers and thus give us the opportunity and foundation to implement *FreeContainer*.

Container Communication. As containers are essentially processes, there are many traditional schemes to optimize inter-process communication network. Shared memory scheme allows containers on the same host to communicate with each other. Overlay routers (and virtual switches) provide the possibility for containers on different hosts to be remote accessed by adopting DPDK [14] or RDMA [13]. These technologies optimize network latency among containers in different ways. *FreeContainer* addresses the communication issues from another perspective. The proposed approach can easily be applied on or combined with the existing techniques.

VM Technologies. As the tussle between communications and isolations is not unique to containers, one may argue that VMs suffer from similar inefficiency and this issue has already been studied using technologies like Network Virtual Machine (NetVM) [24], NetMap [25] and VALE [26]. NetVM provides a shared-memory framework to allow zero-copy delivery of data between VMs. ClickOS [21] adopts VALE instead of Open vSwitch to provide sharing buffers between kernel and userspace to eliminate memory copies. But NetVM requires VMs to be on the same host, so the inter-host communications cannot be handled. NetMap and VALE are sub-optimal to intra-host setting [11] compared with shared-memory. These technologies cannot be used in container systems directly, while our proposed *FreeContainer* retains the advantages of

containerization, for both intra-host and inter-host communications.

III. Background and Overview

In this section, we first give a brief introduction of containerization relevant techniques in Subsection III-A. On this basis, we formulate the optimization problem to be addressed in Subsection III-B.

A. Container Group based Architecture

Containerization is gaining tremendous popularity recently due to its convenience and good performance on deploying applications and services. First, containers provide good isolations by using namespace technologies (e.g., chroot [27]), 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, which make containers highly portable and fast to distribute. Docker [28] is one of the most popular software containerization platforms, which provides much convenience to application developers by enabling containers to run on the top of any infrastructures.

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 cloud or server clusters, and managed by a cluster orchestrator such as Kubernetes [22], Matrix [23] and Mesos [29]. Using name services, these orchestrators can quickly locate all the containers implemented on different hosts, so application upgrade and failure recovery can be well handled. Moreover, it is also easy to build, replace or delete containers.

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 will affect the overall service performance [11]. The above orchestrators provide the possibility to leverage containers, but how to manage container groups for higher throughput and lower latency is still a pending problem.

B. Problem Formulation

In this section, we formally define the container redistribution problem which aims to optimize the overall throughput and container communications. The throughput is measured by the resource utilization cost and residual resource balance cost. Table II defines the notations to be used in the problem definition.

Resource Utilization Cost. If the resource utilization of a host is much higher than others, it will easily become the bottleneck of a chained service, seriously degrading the overall throughput. The ideal situation is that all hosts enjoy equal resource utilizations. Therefore, we define the resource utilization cost as the variance of resource usage of all hosts, i.e.,

TABLE II: Notation Definitions.

Notation	Meaning
$\mathbb{H}$	*The set of physical hosts*
$\mathbb{S}$	*The set of deployed services* $S(i)$
R	*Multiple resources*
$c_{k,i}$	*The* i*th container of the* k*th service*
h_i	*The* i^{th} *host*
$C(h,r)$	*Capacity of resource* r *on host* h
$U(h,r)$	*Usage of resource* r *on host* h
$H(c)$	*The assigned host for container* c
$H'(c)$	*The reassigned host for container* c
D_c^r	*Demand of resource* r *of container* c

$$Ucost = \sum_{r\in R}\sum_{h\in\mathbb{H}} \frac{[U(h,r)-\bar{U}(r)]^2}{\left|\mathbb{H}\right|} \tag{1}$$

where $\bar{U}(r)$ is the average utilization of resource r on all hosts $h \in \mathbb{H}$ and $Ucost$ represents the equalization degree of resource utilization.

Residual Resource Balance Cost. Any amount of CPU resources without any available RAM is useless for future container assignments or coming requests, so the residual amount of multiple resources should be balanced [30]. Let $t(r_i, r_j)$ represent the target proportion between resource r_i and resource r_j. We define the total balance cost as follows:

$$\begin{aligned} Bcost &= \sum_{(r_i,r_j),\forall r_i\neq r_j} cost(r_i,r_j) \\ cost(r_i,r_j) &= \sum_{h_k\in\mathbb{H}} max\{0, A(h_k,r_i) - A(h_k,r_j)\times t(r_i,r_j)\} \end{aligned} \tag{2}$$

where $A(h,r) = C(h,r) - U(h,r)$ refers to the residual available resource r on host h.

Communication Cost. Containers from the same service often exchange control and data messages frequently. As has been mentioned earlier, the communication overhead exists mainly in host networks, while the inner-host communication is negligible compared with network communications. So we leave out the inner-host communication here and only consider the network communication overhead between hosts. For a pair of communication containers $c_i \Leftrightarrow c_j$ instantiated on $H(c_i)$ and $H(c_j)$, let $f(H(c_i), H(c_j))$ represent the network communication overhead between the two hosts. Let $Ccost$ denote the total communication overhead, we have:

$$\begin{aligned} Ccost &= \sum_{S_k\in\mathbb{S}}\sum_{\forall c_i,c_j\in S_k} f(H(c_i),H(c_j)), \\ f(H(c_i),H(c_j)) &= 0, \text{ if } H(c_i)=H(c_j) \text{ or } c_i \nLeftrightarrow c_j. \end{aligned} \tag{3}$$

where $c_i \nLeftrightarrow c_j$ means there is no communication between

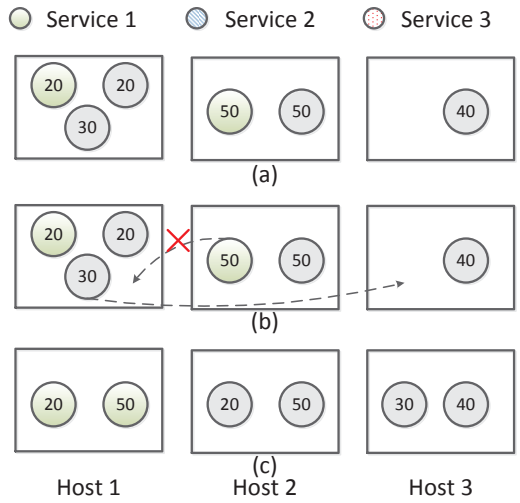


Fig. 2: There are three services each with two containers: (a) shows the current assignment; (b) is the inefficient reassignment; (c) is the best solution.

container c_i and c_j.

Based on the above definitions, the objective function to be minimized can be defined as the weighted sum of all the costs, i.e.,

$$Cost = w_U * Ucost + w_B * Bcost + w_C * Ccost \quad (4)$$

IV. FreeContainer Design

In this section, we present the design of FreeContainer. We first discuss the constraints of the container re-distribution problem and then propose a two-stage container re-distribution algorithm.

A. Constraints and Motivation

1) Constraints: To make sure online services are not interrupted, the container re-distribution process should be transparent. In order to do that, some constraints need to be satisfied during container migrations.

Constraint 1: (Capacity) In each host, resource usage cannot exceed its capacity, i.e.,

$$U(h,r) \leq C(h,r), \forall h \in \mathbb{H}, \forall r \in R \quad (5)$$

To ensure failure recovery, each application has a certain number of duplications. Assume there are $d_{k,i}$ duplicates for container $c_{k,i}$, and $c^m_{k,i}$ indicates the mth duplicate.

Constraint 2: (Duplicate) Any two different duplicates of the same container cannot be assigned to the same host:

$$m \neq n \Rightarrow H(c^m_{k,i}) \neq H(c^n_{k,i}), \forall c_{k,i} \in S_k \quad (6)$$

For service availability purpose, any migrated container cannot be destroyed at the original host until the new instance is built at the new host. Thus, resources are consumed at both original host $H(c)$ and new host $H'(c)$ during container migration, which we call transient property.

Constraint 3: (Transient) Resources are required twice, i.e.,

$$\sum_{H(c)=h_i \bigcup H'(c)=h_i} D^r_c \leq C(h_i, r), \quad \forall c \in S_i \in \mathbb{S}, \forall h_i \in \mathbb{H}, \forall r \in R \quad (7)$$

Besides the above hard constraints, there are also some other requirements that we need to take into consideration.

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 hosts or even different datacenters. As these containers are sensitive to the same resource (CPU-dominant in the web search example), they cannot be put on the same host, otherwise, there will be serious waste of other resources like memory and I/O. Therefore, for each service $S_i \in \mathbb{S}$, let $M(S_i) \in \mathbb{N}$ be the minimum number of different hosts where at least one container of S_i should run, we can define the following spread constraints for each service.:

Constraint 4: (Spread) The containers for S_i should be spread on at least $M(S)$ hosts, i.e.,

$$\sum_{h_i \in \mathbb{H}} min(1, \left| c \in S_i \in \mathbb{S} | H(c) = h_i \right|) \geq M(S_i), \quad \forall S_i \in \mathbb{S} \quad (8)$$

Different functions in one application are usually sensitive to different resources. In the above web service example, while the search module is CPU-dominant, a load balancer is bandwidth-dominant. These two modules need frequent information exchange and data transmissions, so the communication latency among the corresponding containers is crucial to the application performance. In order to mitigate network latency, these associated containers with frequent interactions should be reassigned to the same host. Therefore, we define the following co-locate constraints.

Constraint 5: (Co-locate) Associated containers should be on the same host, i.e.,

$$H(c_{k,i}) = H(c_{k,j}), \quad \text{if } c_{k,i} \text{ and } c_{k,j} \text{ are associated containers in } S_k. \quad (9)$$

2) Example: The container re-distribution problem defined above is a combinational optimization problem, which is NP-complete. Many classical heuristic algorithms have been used to solve similar problems [31]–[33]. However, we shall show that the existing approaches are not efficient to handle big containers at the hot hosts due to transient constraints in our problem.

Consider the example as shown in Figure 2, three services each with two containers are mixed deployed on three hosts. The hosts have a same capacity of 100. The numbers in Figure 2 represents the resource consumptions of the containers. Figure 2(a) shows the current distribution and Figure 2(c) shows the optimal distribution. In order to achieve the optimal distribution, we need to move 30 from host 1 to host 3, and move 50 from host 2 to host 1. However, migrating 50 from

host 2 to host 1 is impossible since the transient constraint will be violated at host 1 (the sum of resource consumption at host 1 will be larger than 100 if 50 is moved in).

In fact, if we first move 30 from host 1 to host 3 and suppose the resource consumed is released at host 1 after migration, then 50 can be moved from host 2 to host 1. Inspired by this observation, we propose *FreeContainer*, which is a two-stage strategy for more efficient container re-distribution.

B. FreeContainer

FreeContainer aims at finding a reassignment solution of containers to minimize the overall cost (Equation 4), *FreeContainer* has two stages, which are $Sweep$ and $Search$. The $Sweep$ stage aims to handle hot hosts, especially for the big containers at the hot hosts. Based on the results generated in the $Sweep$ stage, $Search$ stage adopts a tailored variable neighborhood local search to further optimize the solution.

1) Sweep: As has been shown by the example in Figure 2, the existing solutions for the reassignment problem are not efficient to handle hot hosts, especially for big containers due to transient resource constraints. To address this issue, we propose a novel two step solution in the Sweep stage. As we know, the hosts with high resource utilizations are usually the bottleneck of the overall throughput. To mitigate these bottlenecks, general methods try to move containers away from these hosts, however in this manner, the residual resources at each host might be not sufficient for accommodating big containers in the migrating process.

Different from the existing solutions, $Sweep$ solves the problem from a totally different perspective. In the first step, we try to empty the spare hosts as much as possible by moving out containers from spare hosts to other hosts. This will free up space for accommodating more big containers from hot hosts. In the second step, we handle hot hosts by move containers from hot hosts to spare ones. The pseudo-code of $Sweep$ is shown in Algorithm 1.

We select N "hot" hosts in the host set and clear up N "spare" hosts for them. For each container c_k on the spare host h, $FindPosition$ function tries to find a better position h' for it and we then move c_k from h to h'. After that, spare hosts are set free and can be assigned with containers from hot hosts. Note that the candidate hosts in $FindPosition$ function are the hosts that are neither hot nor spare. After that, we try to eliminate hot hosts by moving out containers to the free hosts, the move action continues until the utilization drops below the mean utilization of all hosts.

The intention of emptying spare hosts in the first step is: we observe that hot hosts often have more big containers which are supposed to be moved to the spare hosts for cost optimization; clearing up spare hosts will make them have more space to accommodate the big containers.

2) Search: After $Sweep$, the communication cost and resource utilization cost may still be high. $Search$ stage will further optimize the solution using a local search algorithm. Specifically, the $Search$ algorithm combines a variable neighborhood search procedure and a tailored move action: 1) The local search procedure explores three kinds of neighbors: shift, swap and replace. In each neighbor set, one of the best three moves is randomly chosen, and this action will be executed if the overall cost is improved; 2) The move action will be executed at the beginning of each search iteration, resorting $\mathbb{H}$, replacing the current state and restarting the search procedure. With this move action, the $Search$ algorithm can escape from local optima.

Algorithm 1 Sweep

```
Sort ℍ in descending order according to CPU utilization
ℍ_hot ← {h_k | U(h_k) > safety threshold}
N = count(ℍ_hot)
Sort ℍ in ascending order according to CPU utilization
ℍ_cold ← {ℍ_1, ..., ℍ_N}
for h_i ∈ ℍ_cold do
    for containers H(c_j) = h_i do
        h' = FindHost(c_j)
        MOVE(c_j, h_i, h')
    end for
end for
for h_i ∈ ℍ_hot do
    while U(h_i) > Ū(h) do
        select c_j (H(c_j) = h_i)
        MOVE(c_j, h_i, ℍ_cold)
    end while
end for
```

Shift. A shift move is to reassign a container from one host to another. The shift neighbor $\mathbb{N}_{shift}$ is defined as the set of reassignment plans that can be achieved by a shift move.

As shown in Figure 3(a), shift move is the most simple neighbor exploration that directly reduces the overall cost. For example, reassigning a container from a "hot" host to a "spare" host will reduce $Ucost$; moving a CPU-intensive container from a host with little residual CPU will reduce $Bcost$; moving a container nearer to its group members reduces $Ccost$.

Swap. A swap move is to exchange the assignment of two containers on different hosts, so the swap neighbor $\mathbb{N}_{swap}$ is defined as the set of reassignment plans that can be achieved by a swap move.

Figure 3(b) shows a simple case of swap move. 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.

Replace. A replace move is more complex than the shift or swap move, which tries to reassign a container from h_A to h_B under the premise to move the zombie containers from h_B to h_C. A zombie container on h_B is a container that was planned to move here (from other hosts) in the earlier search process.

We represent zombie containers with dashed edges. Since the zombie containers have not been migrated, it will not incur additional overhead if we reassign them to other hosts.

See Figure 3(c) for an example, there are three hosts

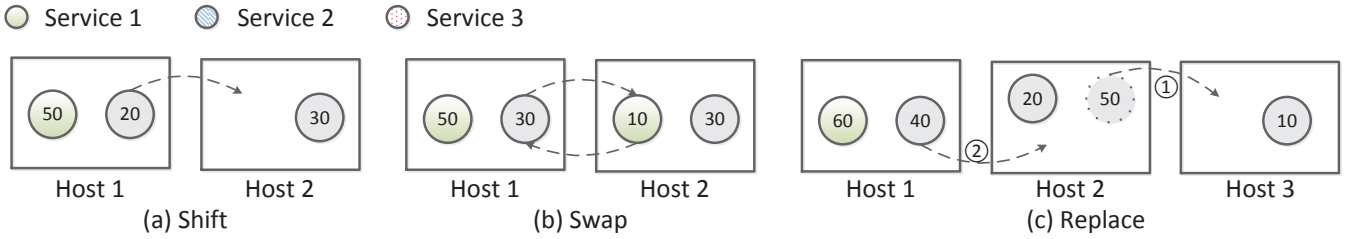


Fig. 3: Three kinds of move neighbors the variable neighborhood search procedure explores.

$(f(h_1, h_2) = f(h_2, h_3) = f(h_1, h_3))$ and the current utilizations are 100%, 70% and 10%, respectively. The average utilization is 60%, so the overall cost is:

$Cost_0 = \frac{40^2+10^2+50^2}{3} + 2 \times f(h_1, h_2) = 1400 + 2f$

With a traditional adjustment scheme, the container of Service 2 on Host 1 will be moved to Host 3 (reduce $Ucost$ and keep $Ccost$ unchanged). Then utilizations become 60%, 70% and 50%, so,

$Cost_1 = \frac{10^2+20^2+10^2}{3} + 2 \times f(h_2, h_3) = 200 + 2f$

Replace moves the zombie container on Host 2 to Host 3, and this makes it possible to move the container of Service 2 from Host 1. The final utilizations for Hosts 1, 2, 3 are all 60% and the overall cost is:

$Cost_2 = 0 + 2 \times f(h_2, h_3) = 2f$

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 scenarios 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 branches 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 *FreeContainer* is linear to o_s.

The pseudo-code of *Search* is shown in Algorithm 2. The tailored move action resorts all the hosts and updates the current state in Line 5. This procedure is necessary to our algorithm because it helps the neighborhood search procedure escape from local optima. In the following search procedure, we select 2δ hosts as the set of candidates N from the top δ and tail δ hosts. We leverage neighbor searching only on these candidate hosts since they play more important roles in cost reduction. The three search actions in Line 8, 12 and 16 explore shift, swap and replace neighbors, respectively. Each neighborhood search returns the best three solutions. We randomly select one and compare it with the current state. If the cost is improved, we accept this action. The above iteration continues until the overall cost falls to the pre-set threshold T.

With *Sweep* and *Search*, *FreeContainer* successfully improves system throughput and reduces network communication overhead by solving the "hot" host problem and exploring better reassignment plans for container groups. In Section V, we'll give a detailed algorithm analysis and prove that the deviation between *FreeContainer* and the theoretical optimal solution has an upper bound.

Algorithm 2 Search

1: Initialize the starting point P_{start}
2: $P_{crt} \leftarrow P_{start}$
3: $P_{best} \leftarrow P_{start}$
4: **while** $Cost(P_{crt}) > T$ **do**
5: Sort $\mathbb{H}$ by resource utilization
6: Update P_{crt}
7: $N \leftarrow \mathbb{H}(Top(\delta)) + \mathbb{H}(Tail(\delta))$
8: $P_{shift} \leftarrow$ shiftSearch(P_{crt}, N)
9: **if** $Cost(P_{shift}) < Cost(P_{crt})$ **then**
10: $P_{crt} \leftarrow P_{shift}$
11: **end if**
12: $P_{swap} \leftarrow$ swapSearch(P_{crt}, N)
13: **if** $Cost(P_{swap}) < Cost(P_{crt})$ **then**
14: $P_{crt} \leftarrow P_{swap}$
15: **end if**
16: $P_{replace} \leftarrow$ replaceSearch(P_{crt}, N)
17: **if** $Cost(P_{replace}) < Cost(P_{crt})$ **then**
18: $P_{crt} \leftarrow P_{replace}$
19: **end if**
20: $P_{best} \leftarrow P_{crt}$
21: **end while**
22: Output P_{best}

V. Algorithm Analysis

In this section, we would like to prove that the output of *FreeContainer* P_{best} 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 [34]. More specifically, this $(1+\epsilon, \theta)$-approximation can be formulated as the following inequality:

$$Pr[|\hat{P} - P^*| \leq \epsilon P^*] \geq 1 - \theta \tag{10}$$

if $\epsilon = 0.05$ and $\theta = 0.1$, it means that the output of *FreeContainer* $\hat{P}$ 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 $\mathbb{H}$ to run *Search*, the other is the stopping condition. Specifically, the core idea of the *search* algorithm is to select some candidate hosts from $\mathbb{H}$, 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 *FreeContainer* $\hat{P}$ 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.

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 $\frac{\epsilon}{2}$.

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:

$$Pr[|V - E(V)| \geq \varepsilon] \leq e^{-2\varepsilon^2 k} \tag{11}$$

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:

$$E(\hat{Q}) = Y < Y^{\frac{n}{2\delta}} \tag{12}$$

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:

TABLE III: Server Configuration.

CPU		Memory		SSD	
Capacity	No.	Capacity	No.	Capacity	No.
1	37	1	1970	1	4
0.8	1819	0.75	24	0.3	309
0.55	3210	0.5	2437	0.18	2433
0.45	814	0.35	1578	0.1	1567

$$E(Q^*) = (1 - \frac{x_i}{b-a})^n = E(Y_i)^n = E(Y)^{\frac{n}{2\delta}} \tag{13}$$

and

$$Pr[|E(\hat{Q}) - E(Q^*)| \geq \frac{\epsilon}{2}] < Pr[|Y^{\frac{n}{2\delta}} - E(Y)^{\frac{n}{2\delta}}| \geq \frac{\epsilon}{2}] \tag{14}$$

Through the above Hoeffding Bound (Inequation 11), we know that:

$$Pr[|Y^{\frac{n}{2\delta}} - E(Y)^{\frac{n}{2\delta}}| \geq \frac{\epsilon}{2}] \leq e^{-2(\frac{\epsilon}{2})^2 * 2\delta} \tag{15}$$

Finally,

$$Pr[|E(\hat{Q}) - E(Q^*)| \geq \frac{\epsilon}{2}] \leq e^{-\epsilon^2 \delta} \tag{16}$$

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, *FreeContainer* 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 in Line 7 of Algorithm 2.

VI. Implementation

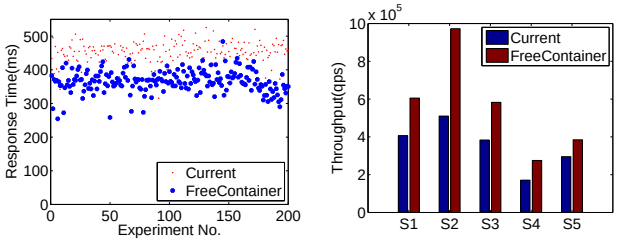
We built *FreeContainer* in the cluster orchestrator system that manages virtualized services of Baidu Company. The implementation environment is described as follows:

Server Cluster. We implement *FreeContainer* in an IDC with about 6000 servers spreading across several geographical cities and placed in different regions. The capacities of servers are shown in Table III after normalization. Specifically, we take CPU as an example. There are 37 servers in top configuration, and we normalize the top configuration to 1. Thus, 1819 servers are in 80% of the top configuration. We leave out some configurations with small quantities of servers and just list the main configurations.

Deployed Services. In this IDC, 35 services are deployed together with some other background services. Each service has a number of duplications, and each duplication consists of a group of containers that would communicate with each other.

TABLE IV: Service Information.

Service	Duplication No.	Containers/Dupl
S_1	1	3052
S_2	6	378
S_3	10	96
S_4	3	192
S_5	6	98



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

Fig. 4: System performance before and after deploying *FreeContainer*.

Table IV shows the detailed information of 5 representative selected services. Taking S_2 as an example, there are 6 service duplications and each of them has 378 containers that would communicate with each other. S_1 represents the services with a large amount of distinct containers and S_3 represents the services with multiple duplications. *FreeContainer* aims at minimizing the communication latency within the same service and improving system throughput simultaneously.

Comparison System. We also implemented two other algorithms to compare. 1) The first one adopts a greedy algorithm. In each iteration, it tries to move containers from the "hottest" host to the "sparest" one. This algorithm reduces $Ucost$ directly in a straightforward way. 2) The second is noisy local search method (NLS), which is based on the winner team solution for Google Machine Reassignment problem (GMRP) [30]. This method reallocates processes among a set of machines to improve the overall efficiency.

VII. Evaluation

We have conducted extensive experiments with variant parameter settings in Baidu's IDC to evaluate *FreeContainer* from different aspects: 1. the efficiency and advantages of *FreeContainer*; 2. the comparison results between *FreeContainer* and the other two systems; 3. the algorithm efficiency under different parameter settings.

A. FreeContainer Performance

We implement *FreeContainer* in IDC with 6000 servers described in the above section. To evaluate algorithm performance on container communication and the overall cost, we conduct a series of experiments to measure the following system features: service span, request response time, cluster throughput and resource utilization.

Response Time. To show the efficiency on communication cost reduction, we measure service response time, which 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 4(a), where the small red (big blue) points denote the request response time before (after) deploying *FreeContainer*. The average response times are 451 ms and 365 ms, respectively. From this figure, we can conclude that *FreeContainer* reduces up to 20% communication latency.

Cluster Throughput. For container-based network functions, throughput is a key indicator of performance. We perform pressure measurement on this cluster and the service throughput is shown in Figure 4(b). We still choose the 5 representative services and show the throughput before and after deploying *FreeContainer*. Taking S_2 as an example, the maximum throughput before deploying *FreeContainer* is 510008 qps and raises to 972581 qps after implementing our algorithm, getting an increase of 90%. But different services enjoy different optimization. For S_5, the throughput is improved from 295581 qps to 384761 qps, with 30% increase. This is because there are only 588 interactive containers in S_5 but 2268 containers in S_2. These results show that for a particular service, the more containers in a communication group, the higher efficiency *FreeContainer* obtains.

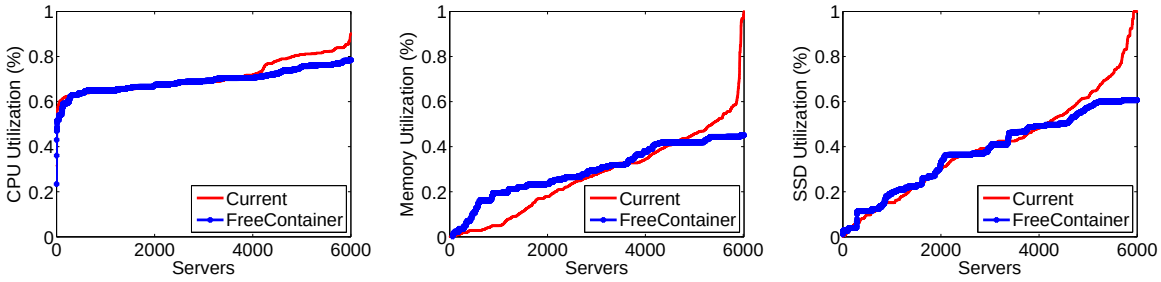
Resource Utilization. Resource utilization is another indicator of the performance. If the resource utilizations are balanced among hosts, the throughput is generally also good. We measure resource utilizations under pressure experiments and show the cumulative distribution function (CDF) results in Figure 5 (CPU in Figure 5(a), memory in Figure 5(b) and SSD in Figure 5(c)). From these figures we can see that *FreeContainer* eliminates the long tails of resource utilizations. Taking CPU utilization as an example, there are about 800 servers whose utilization exceeds 80%. These servers suffer from long request latency due to CPU resource shortage, which are the bottlenecks of the overall network functions. From the results we can see, *FreeContainer* obtains more balanced resource utilizations.

B. Comparison with other algorithms

To compare *FreeContainer* with the state-of-the-art solutions, we build two comparison systems – Greedy and NLS. As we cannot deploy these comparison systems in the real IDC for safety concerns, we implement them in a cluster with 2513 servers.

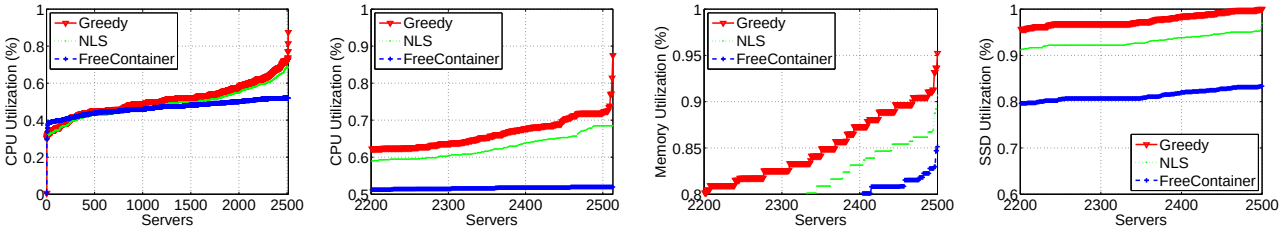
As illustrated in the above subsection, server resource utilizations reflect the cluster performance under pressure experiment, i.e., when burst occurs, the more balanced server resource utilizations are, the larger throughput a cluster can obtain. Along this way, we measure resource utilizations in this experimental cluster and show the results in Figure 6.

Figure 6(a) shows the CDF of CPU utilizations of those 2513 servers. There are about 330 servers whose CPU utilizations exceed 60% under the greedy algorithm and 210 servers under the NLS algorithm. But when leveraging *FreeContainer*,



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

Fig. 5: Resource utilizations of the Baidu server cluster before and after deploying *FreeContainer*.



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

Fig. 6: The comparison of resource utilizations of different systems.

TABLE V: Average CPU Utilization of Bottleneck Servers.

Algorithm	**Top 1%**	**Top 5%**	**Top 10%**
Greedy	0.7362	0.7033	0.6749
NLS	0.7190	0.6919	0.6566
FreeContainer	0.5714	0.5703	0.5687

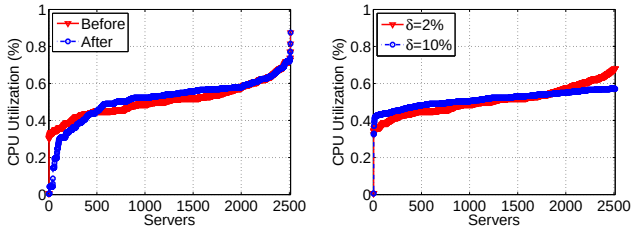
the highest CPU utilization falls down to 52%, which is much better than greedy and NLS.

For high performance network services, the overall throughput of the system is generally determined by the hot hosts. We collect the resource usages of the top 300 hot servers produced by each algorithm, and the results are shown in Figure 6. Taking SSD as an example, the average utilizations of the top 300 hot servers under greedy, NSL and *FreeContainer* are 97.71%, 93.15% and 81.33%, respectively. To clearly show the quantified optimization results, the average CPU utilizations of top hot servers are shown in Table V. The overall average CPU utilization of the 2513 servers is 51.16%. We can see that FreeContainer's performance is very close to the lower bound, and outperforms greedy and NSL by up to 70%.

We attribute this to: First, we take $Ucost$ and $Bcost$ into consideration to minimize the difference in resource utilizations and balance residual multiple resources. Second, the $Sweep$ stage in *FreeContainer* makes room for the following search procedure, based on which $Search$ stage could explore more branches to find better solutions.

C. Algorithm Efficiency

In this subsection, we evaluate the impacts of different parameter settings on the performance of *FreeContainer*.



(a) The efficiency of *sweep*. (b) Different $delta$.

Fig. 7: Algorithm efficiency with different configurations.

Figure 7(a) shows the resource utilization of the 2513 servers before and after running the *Sweep* algorithm. As we described in Section IV, the objective of the *Sweep* stage is to make as much room as possible for the *Search* stage by freeing up a group of servers and using these servers to instantiate containers that are moved from "hot hosts".

Figure 7(b) shows the results under different exploring scopes. $\delta = 2\%(10\%)$ means that in each exploring iteration, we select $4\%(20\%)$ servers as the set of candidates from the top $2\%(10\%)$ and the tail $2\%(10\%)$ and leverage neighbor searching on these candidate servers. As can be seen from Figure 7, larger δ achieves more balanced resource utilizations, which confirms the analysis given in Section V.

In summary, *FreeContainer* manages container groups deployed in clusters in a high-efficient way in terms of minimizing communication overhead among interactive containers and balancing server resource utilizations. The experimental results on Baidu clusters show that *FreeContainer* increases

the overall throughput up to 90% at most.

VIII. Conclusion

Although containerization provides good isolation and portability for network functions, the container communication is threatened by the requirements from high-performance VNFs. To address this problem, the existing network solutions either make hardware modification or sacrifice some system performance, making them too costly and undermine the advantages of containerization.

In this paper, we proposed *FreeContainer*, which is a communication-aware container re-distribution method that manages container groups for high-performance VNFs. *FreeContainer* requires no hardware modification and is completely transparent to online services and introduces no disruption. We implemented *FreeContainer* in Baidu clusters with 6000 servers and 35 services and conduct a series of experiments under pressure testing. The measurements from real network indicate that *FreeContainer* can: 1) reduce the communication overhead among interactive containers by reassigning containers; 2) increase the overall throughput up to 90% in terms of balancing resource utilizations; 3) outperform the state-of-the-art solutions up to 70%.

Acknowledgments

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

This work was supported by the National Natural Foundation of China (61472212), National Science and Technology Major Project of China (2015ZX03003004), the National High Technology Research and Development Program of China (863 Program) (2013AA013302, 2015AA015601), EU Marie Curie Actions CROWN (FP7-PEOPLE-2013-IRSES-610524). Yusen Li's work was supported in part by the NSF of China under grant 61602266, and NSF of Tianjin under grant 16JCYBJC41900. Dan Wang's work was supported in part by PolyU G-YBAG.

Corresponding authors are Yusen Li, Ke Xu and Dan Wang.

References

[1] [online], "Control groups definition, implementation details, examples and api." https://android.googlesource.com/kernel/msm/+/android-wear-5.0.2_r0.1/Documentation/cgroups/00-INDEX, 2016.

[2] E. W. Biederman and L. Networx, "Multiple instances of the global linux namespaces," in *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2006, pp. 101–112.

[3] "Openstack," https://www.openstack.org/, 2016.

[4] "Openvz linux containers." https://openvz.org/Main_Page, 2016.

[5] "Freebsd - the power to serve." https://www.freebsd.org/, 2016.

[6] "Oracle solaris containers." http://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html, 2016.

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

[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, 2007, pp. 225–230.

[9] M. Corporation, "Build your future with windows server 2016," https://www.microsoft.com/en-us/cloud-platform/windows-server, 2016.

[10] VMware, "Vmware virtualization software for desktops, servers and virtual machines for public and private cloud solutions." http://www.vmware.com, 2016.

[11] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar, "Freeflow: High performance container networking," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 2016, pp. 43–49.

[12] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[13] Mellanox, "Rdma aware networks programming user manual." http://www.mellanox.com/, 2016.

[14] "Data plane development kit (dpdk)." http://dpdk.org/, 2016.

[15] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *OSDI*, vol. 10, no. 1, 2010, p. 24.

[16] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 583–598.

[17] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 1, pp. 97–107, 2014.

[18] "Baidu," http://www.baidu.com.

[19] K. Xu, T. Li, H. Wang, H. Li, Z. Wei, J. Liu, and S. Lin, "Modeling, analysis, and implementation of universal acceleration platform across online video sharing sites," *IEEE Transactions on Services Computing*, 2016.

[20] Y. Zhang, K. Xu, G. Yao, M. Zhang, and X. Nie, "Piebridge: A cross-dr scale large data transmission scheduling system," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 553–554.

[21] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014, pp. 459–473.

[22] "Kubernetes." http://kubernetes.io/, 2016.

[23] "Matrix." http://wiki.baidu.com/display/solaria/Solaria, 2016.

[24] J. Hwang, K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[25] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[26] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 61–72.

[27] "Freebds.chrootcfreebdsmanpages." http://www.freebsd.org/cgi/man.cgi, 2016.

[28] "Docker." http://www.docker.com/, 2016.

[29] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[30] H. Gavranović and M. Buljubašić, "An efficient local search with noising strategy for google machine reassignment problem," *Annals of Operations Research*, pp. 1–13, 2014.

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

[32] J. A. Dıaz and E. Fernández, "A tabu search heuristic for the generalized assignment problem," *European Journal of Operational Research*, vol. 132, no. 1, pp. 22–38, 2001.

[33] R. Masson, T. Vidal, J. Michallet, P. H. V. Penna, V. Petrucci, A. Subramanian, and H. Dubedout, "An iterated local search heuristic for multi-capacity bin packing and machine reassignment problems," *Expert Systems with Applications*, vol. 40, no. 13, pp. 5266–5275, 2013.

[34] M. H. Zhu Han and D. Wang, *Signal processing and networking for big data applications*. Cambridge Press, 2017.