Maximizing Container-based Network Isolation in Parallel Computing Clusters

Shiyao Ma*, Jingjie Jiang*, Bo Li*, and Baochun Li†
*Department of Computer Science and Engineering, Hong Kong University of Science and Technology
†Department of Electrical and Computer Engineering, University of Toronto

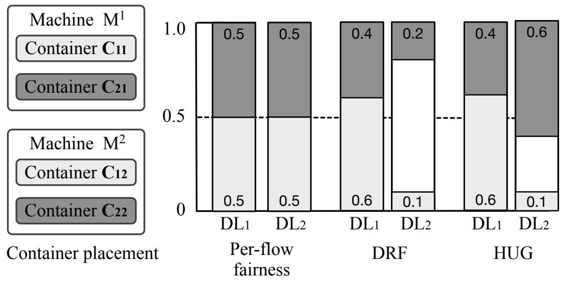
***Abstract*—Data-parallel applications, especially those associated with user-facing web services, have struggled to enhance their worst case performance. It is therefore important to improve the minimum amount of resources guaranteed for applications in a cluster. Existing cluster management frameworks, however, provide isolation for computation resources (such as CPU) only, and are oblivious to network isolation guarantees. In this paper, we design, implement and evaluate *Libra*, a new cluster management framework that helps to maximize the isolation guarantee for the bandwidth requirements from applications. We start with a theoretical analysis of the network sharing problem, which contains two key steps: container placement and bandwidth allocation. By collecting the status of access links and the bandwidth demand of applications, we coordinate the placement of containers to minimize the system bottleneck such that the bandwidth guarantee for applications can be optimized. We further embrace host-based rate limiting to ensure such maximized bandwidth guarantee can be reached without hurting network utilization. Both our testbed-based experiments and large-scale simulations demonstrate that *Libra* significantly improves the network isolation guarantee: in comparison with existing cluster managers and network schedulers, the performance gain is more than 105.59%. Meanwhile, it improves application performance by 57.71% and maintains high network utilization.**

I. Introduction

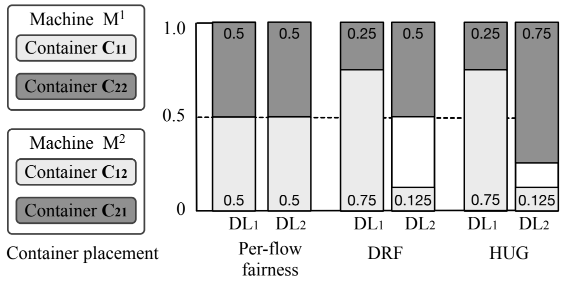
Currently, cluster managers from both the academia and industry are following the trend of *resource isolation*. Mesos [1] and the cluster management schemes used in production clouds [2] [3] support isolating computation resources (*i.e.*, CPU and memory) and storage resources (*i.e.*, disk I/O) via lightweight containers [4]. Applications can independently schedule their tasks onto the containers allocated by the cluster manager. Such containerized deployment guarantees that tasks from different applications are securely isolated from each other. To provide consistent and predictable performance, the cluster manager guarantees the amount of aggregated computation resources that tasks in a container can access.

On the other hand, while network resources are crucial for application performance, they are not well orchestrated or isolated. The cluster manager can only set a *maximum* limit to the amount of bandwidth and the number of ports each container can use. Such a restriction only prevents containers from exhausting network ports or consuming too much bandwidth [1]. Rather than maximum bandwidth, it is commonly acknowledged [5], [6], [7] that applications expect guaranteed *minimum* bandwidth, *i.e.*, an *isolation guarantee*. If it is not provided, applications are not able to estimate the worst-time completion time of the tasks involving network transfers [8]. Such a lack of predictability negatively affects scheduling efficiency and application performance since the actually achieved bandwidth can be arbitrarily small.

The research was supported in part by grants from RGC under the contracts 615613 and 16211715, a grant from NSFC and Guangdong joint project under the contract U1301253.



(a) The isolation guarantees achieved are 0.417, 0.5 and 0.5.



(b) The isolation guarantees achieved are 0.417, 0.625 and 0.625.

Fig. 1. A motivating example: an application A_1 requests for c_{11} and c_{12}, while A_2 desires c_{21} and c_{22}. Placement of containers in (a) and (b) lead to different isolation guarantee and utilization of downlinks (DLs) with the same set of bandwidth allocation schemes.

As illustrated in a recent work on bandwidth allocation schemes (*i.e.*, HUG [5]), optimal shuffle schedules, long-running services and real-time streaming applications exhibit correlated demands on network resources of their constituent containers. The isolation guarantee to an application is thus defined as the minimum *demand-normalized* rate allocation over all of its containers. For instance, in Fig. 1, an application A_1 needs two containers c_{11} and c_{12} with demands for downlink bandwidth equal to 1.2 and 0.2 Gbps respectively, while A_2 needs c_{21} and c_{22} with 0.8 and 0.4 Gbps downlink bandwidth.

Both machines have 1 Gbps downlink. As shown in Fig. 1(a), the isolation guarantee that A_1 achieves under DRF (dominant resource fairness) [9] or HUG is $\min(\frac{0.6}{1.2}, \frac{0.1}{0.2}) = 0.5$.

However, the schemes mentioned in the example as well as other solutions [6], [7] only restricted themselves to bandwidth allocation after containers are launched for applications. In essence, how to place the containers for concurrent applications determines the best isolation guarantee that any bandwidth allocation scheme can achieve. As shown in Fig. 1(b), by changing the placement of c_{21} and c_{22}, the isolation of A_1 and A_2 under HUG [5] and DRF [9] both increase to $\frac{0.75}{1.2}$ or $\frac{0.5}{0.8} = 0.625$. Besides, even without allocating the spared bandwidth, DRF can achieve better link utilization as well.

In this paper, we propose to maximize the minimum bandwidth guarantee that applications can achieve through isolation-aware container placement and bandwidth allocation under a unifying framework. In a multi-tenancy cluster, it is non-trivial to handle the possibly conflicted demands from concurrent applications. On one hand, inter-application fairness is obligatory for cluster management [9]. On the other hand, cluster operators strive for high utilization of resources. Admission control mechanisms [33] that sacrifice network efficiency for isolation guarantee are thus undesirable. Finally, the cluster manager must promptly react to dynamic application demands and fluctuating network conditions to ensure applications can always achieve desirable performance.

Network sharing consists of two inter-dependent steps. The containers first have to be launched onto machines based on their computation and network demands. The flows of the containers then need to be rate limited, such that isolation guarantees both between and within applications can be achieved. Through theoretical analyses within both steps, we propose a container placement algorithm that finds the local optimal solution and a rate limiting algorithm that reaches the isolation guarantee derived in the first stage.

Our major contributions lie in the fact that we reinforce existing cluster management systems by integrating isolation-aware container placement, and improve the optimal network isolation that any bandwidth allocation mechanisms can achieve. By formulating the container placement into a *bottleneck generalized assignment problem* [10], we have found that the key to optimal isolation guarantee is to minimize the normalized system bottleneck. Through combining isolation-aware container placement and bandwidth allocation, we effectively maximize the isolation guarantee of applications and obtain inter-application fairness without sacrificing link utilization. Our extensive experimental results demonstrate the effectiveness of *Libra*: compared to the state-of-the-art bandwidth allocation scheme (*i.e.*, HUG [5]) on top of a network-unaware cluster manager (*i.e.*, YARN [2]), *Libra* improves network isolation guarantee by 17.88% - 139.80%, and improves application performance by up to 68.05%. *Libra* can be automatically deployed using our customized plugins, ready to be used in production clusters.

II. Background and Motivation

A. Network Sharing in the Cloud

According to the statistics collected in production data centers [7], core networks seldom experience severe or persistent congestion, whereas network edges are often fully occupied. Given such observations, we suppose congestions only occur at network edges for simplicity. The network resources that we care about are thus restricted to the bandwidth of access links [5], [7]. Specific network topologies (e.g., [11], [12]) or routing protocols [13], [14] have no influence on the network sharing problem.

Container-based resource sharing can perform either in an offer-oriented [1], [3] or a request-based manner [2]. Although the former is simple and flexible, the cluster manager might be frequently rejected by applications due to offering the undesired containers. On the other hand, request-driven cluster managers are more efficient by allowing applications to specify the preferred locations, relative priorities and resource requirements of their containers. Our proposal is based on the request-driven resource sharing model by further enabling applications to express their demands for network bandwidth on access links.

As a datacenter network involves uplinks and downlink on multiple machines, the bandwidth demands of an application are often *correlated* and *elastic* [5], [15] across these links.

Specifically, an application A_i requests multiple containers c_{ij}, each of which is associated with a demand vector $\mathbf{v}_{ij} = (c_{ij}, r_{ij}, d_{ij}, u_{ij})$, representing the demand for CPU, RAM, downlink bandwidth and uplink bandwidth. For a machine M^k with c^k CPU cores, r^k memory space, d^k downlink bandwidth and u^k uplink bandwidth, it is able to launch c_{ij} if its available computation resources are sufficient. However, the bandwidth demands are *elastic* in that the containers can be launched onto a machine even if its available bandwidth is less than the demand. On the other hand, if there is still bandwidth left idle after allocation, the containers can use more bandwidth than its original demand. Apart from such elasticity, the bandwidth demands of containers within an application are also *correlated*: for every d_{ij} bits c_{ij} sends, another container $c_{ij'}$ of A_i should at least send $d_{ij'}$ bits.

To handle the heterogeneous capacities of machines, we further define a machine-specific demand vector $\mathbf{v}^k_{ij}$ for each container. The element of $\mathbf{v}^k_{ij}$ is *normalized* through dividing the corresponding element in $\mathbf{v}_{ij}$ by the relevant capacity of M^k. The network isolation guarantee of an application can then be formally defined by the minimum *demand-normalized* share of its constituent containers:

$$G_i = \min_{c_{ij} \in A_i} \min(\frac{\alpha^k_{ij}}{u^k_{ij}}, \frac{\beta^k_{ij}}{d^k_{ij}}) \quad (1)$$

where α^k_{ij} and β^k_{ij} are the *bandwidth share* allocated to c_{ij} on the uplink and downlink bandwidth of machine M^k. Intuitively, the isolation guarantee of an application captures its progress and predicts its performance. Table I summarizes our notations and definitions.

TABLE I
SUMMARY OF NOTATIONS AND DEFINITIONS.

Notation	Definition
$\mathcal{B}$	the aggregated demands of the system bottleneck
$A_i = \{c_{ij}\}$	an application consisting of multiple containers
G_i	the network isolation guarantee that A_i achieves
ω_i	the weight assigned to A_i based on its category
M^k	a physical machine with different amount of resources (c^k, r^k, d^k, u^k)
U^k, D^k	the aggregated demands for uplink and downlink bandwidth of M^k
$\mathbf{v}_{ij}, \mathbf{v}^k_{ij}$	c_{ij}'s absolute and machine-normalized demand vectors
u_{ij}, u^k_{ij}	the absolute and machine-normalized demand for uplink bandwidth of c_{ij}
d_{ij}, d^k_{ij}	the absolute and machine-normalized demand for downlink bandwidth of c_{ij}
x^k_{ij}	a 0-1 variable indicating whether c_{ij} is placed onto machine M^k
$\alpha^k_{ij}, \beta^k_{ij}$	the machine-normalized allocation for c_{ij} on machine M^k (corresponding to uplink and downlink)
γ^k_{ij}	the actual consumption of bandwidth by flows of c_{ij}

B. Placement of Containers Matters

To optimize the minimum isolation guarantee, we need to consider two indispensable procedures of network sharing. Firstly, the cluster manager needs to decide where to place the containers required by each application. Secondly, the cluster manager should restrict the bandwidth used by each container to achieve the optimal isolation guarantee under the given container placement.

Although some efforts have been made [5], [6], [7] to improve the second phase, the isolation-aware container placement is not well investigated. In essence, container placement determines the optimal network isolation that any bandwidth allocation strategy [5], [6], [7] can achieve. To better understand why this is the case, let's give a comprehensive analysis on the example in Fig. 1.

The placement in Fig. 1(a) launches two containers c_{11} and c_{21} on M^1, and launches containers c_{12} and c_{22} on M^2. Since the bandwidth on each link can be viewed as a type of resources, DL$_1$ is the dominant resource for both A_1 and A_2. To achieve dominant resource fairness [9], c_{11} and c_{21} proportionally share the bandwidth of DL$_1$ based on their demands. As a result, c_{11} and c_{21} get allocated $0.6/1.2$ $(0.4/0.8)$ $= 0.5$ of their demands. The containers using DL$_2$ should also be allocated with 0.5 of their demands as restricted by DRF. Even if the bandwidth allocation strategy in HUG [5] can improve link utilization through re-allocating the unallocated bandwidth on DL$_2$, the isolation guarantee is still restricted by the share c_{11} and c_{21} can get. In contrast, if we launch c_{11} and c_{22} on the first machine, and launch c_{21} and c_{12} on the second machine, every container can get allocated $0.75/1.2 = 0.625$ of its demand. Meanwhile, the link utilizations achieved by strategy-proof strategies [5], [9] are also improved before applying additional backfilling.

From this example we have three important observations. Firstly, the optimal isolation guarantee that an bandwidth allocation strategy can achieve is determined by the placement of containers for concurrent applications. Secondly, although inter-application fairness is beneficial, it is unnecessary to force every container within a single application gets allocated with an equal portion of its demand. A strategy is thus needed to allocate the spare bandwidth to improve link utilization. Most importantly, this example implies that the key to maximum isolation guarantee is to minimizing the system bottleneck link, whose aggregated bandwidth demand is the maximum across the network. We next theoretically analyze the network sharing problem in Sec. III.

III. EFFICIENT NETWORK ISOLATION

In this section, we mathematically formulate the two-stage network sharing problem as two optimization problems. We then propose efficient algorithms to place containers and allocate bandwidth separately.

A. Problem Formulation

As stated in Sec. II, the network sharing consist of two inter-dependent phases: placement of containers and allocation of link bandwidth. On one hand, the placement of containers determines the best guarantee that the subsequent bandwidth allocation strategy can achieve. On the other hand, the actual guarantee achieved helps to evaluate the placement strategy.

We define two set of variables U^k and D^k, which represent the aggregated demands for uplink and downlink bandwidth of each machine:

$$U^k = \sum_i \sum_j \omega_i u^k_{ij} x^k_{ij}, \quad D^k = \sum_i \sum_j \omega_i d^k_{ij} x^k_{ij} \tag{2}$$

where x^k_{ij} is a 0-1 variable to indicate whether a container c_{ij} is launched on M^k. $\omega_i \in [0, 1]$ is the weight assigned to A_i to differentiate among various types of applications. Based on such definitions, we revisit the example in Fig. 1, and calculate the demands for downlinks of the two machines. As shown in Fig. 2, different placements of containers lead to different machine-specific demands. Combined with the isolation guarantee achieved with two placement plans, we find that a more balanced distribution (on the right) of network demands lead to better network isolation for both applications (0.625 as opposed to 0.5).

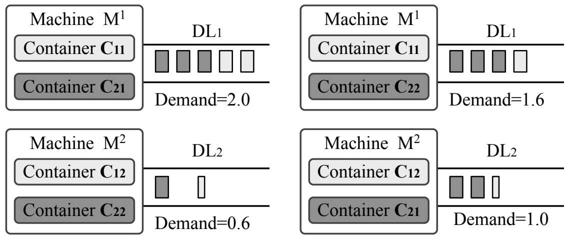


Fig. 2. Different placements of containers may lead to different network status ($\omega_1 = \omega_2 = 1$): the distribution of machine-specific demands is crucial.

Indeed, if we redeem the bandwidth of each link as a type of resources, each application should satisfy the following constraint to achieve dominant resource fairness [5], [9]:

$$G_i = \frac{\omega_i}{\max\limits_k \max(U^k, D^k)} \tag{3}$$

To maximize such isolation guarantee, we need to minimize the load of the bottleneck link $k^* = \arg\max_k(\max(U^k, D^k))$. This coincides with our previous observation from the example. However, the seemingly simple objective turns out hard to achieve given the heterogeneous capacities of machines in the cluster. We formulate the container placement problem with the objective of minimizing the system bottleneck as below:

$$\text{minimize} \quad \mathcal{B} \tag{4}$$

Subject to:

$$\sum_{i,j} u_{ij}^k x_{ij}^k \le \mathcal{B}, \quad \sum_{i,j} d_{ij}^k x_{ij}^k \le \mathcal{B}, \quad \forall k \tag{4a}$$

$$\sum_{i,j} r_{ij}^k x_{ij}^k \le 1, \quad \sum_{i,j} c_{ij}^k x_{ij}^k \le 1, \quad \forall k \tag{4b}$$

$$\sum\nolimits_k x_{ij}^k = 1 \quad \forall i, j \tag{4c}$$

$$x_{ij}^k \in \{0, 1\} \tag{4d}$$

However, maximizing the upper bound of isolation guarantee is incomplete, without per-host rate limiting, all the flows go through the same link will still share the available bandwidth equally. We need to further embrace an allocation strategy to determine the bandwidth that each container can use. The objective is to achieve the best isolation guarantee we derived from the container placement problem, while try to improve link utilization as much as possible. The bandwidth allocation problem is stated below

$$\text{maximize} \sum_{i,j,k} \gamma_{ij}^k \tag{5}$$

Subject to:

$$\sum_{i,j} \alpha_{ij}^k \le 1, \quad \sum_{i,j} \beta_{ij}^k \le 1, \quad \sum_{i,j} \gamma_{ij}^k \le 1, \quad \forall k \tag{5a}$$

$$G_i = \frac{\omega_i}{\mathcal{B}^*} \qquad \alpha_{ij}^k, \beta_{ij}^k, \gamma_{ij}^k \in (0, 1] \tag{5b}$$

γ_{ij}^k represents the actual consumption of bandwidth by flows from the container c_{ij}. We next illustrate our solution to the two-stage network sharing problem.

B. Network Sharing Strategies

It is easy to see the container placement problem stated in Eq. (4) is intractable due to the binary variables x_{ij}^k. Indeed, if we regard each machine as an agent and each container as a task, the bandwidth demand of a container can then be viewed as the processing time an agent needs to process the task. The problem in Eq. (4) is essentially a *multi-resource bottleneck generalized assignment problem* [10], [16]. Although a branch-and-bound algorithm exists to find the optimal solution [16], it calls for exponential computation time even if we try to reach an α-approximate solution (α infinitely approaches 1).

A workaround is to solve the Linear Programming (LP) counterpart of the container placement problem by relaxing the integral constraints in Eq. (4d). Base on this fractional solution, we can further apply rounding and Tabu search [17] to find feasible solutions within several percent of the optimal solution [16]. Suppose n is the total number of containers desired by applications and m is the number of machines in the cluster. The LP relaxation problem has $n+4m$ constraints and nm variables. The complexity of solving the LP relaxation problem is $\mathcal{O}(n^{3.5}n + nm^{3.5})$. In a large cluster, solving such a global problem can still be very time-consuming.

The good news is that n is small enough in practice, given that applications do not come at the same time but one after another. Unfortunately, there are still too many machines to choose from. To reduce the overhead needed to place containers, we select a subset of machines with the most remaining bandwidth as the *candidate* machines for the incoming application. We can then solve the container placement problem in this reduced scope. The size of the candidate set is determined by the number of containers needed by the incoming application. We evaluate the influence of the size of candidate sets in Sec. V-A, and show that the performance degradation is very small since the LP solver can find the *local optimal* solution to our mixed-integer programming problem in a very short time.

After the containers are launched for applications, we can easily calculate the system bottleneck and determine the minimum allocation of each application based on Eq. 3. The basic allocation for each container is thus $G_i d_{ij}^k$ (or $G_i u_{ij}^k$). To maximize link utilization, we further allocate the remaining bandwidth on each link to relevant containers and ensure such backfilled bandwidth is still proportional to the weighted demands. The detailed container placement and allocation algorithms are described in Sec. IV.

IV. Design

In this section, we first present *Libra*'s architecture and its design choices, and illustrate how *Libra* functions in a cluster with dynamic arrivals of applications and elastic demands (Alg. 1). We then discuss the details involved in container placement to minimize the network bottleneck across the cluster (Alg. 2), such that the optimal isolation guarantee achieved through rate limiting (Alg. 3) is maximized.

A. Architectural Overview

Libra fits into request-based cluster management frameworks [2] by further integrating network demands in the resource requests. We only provide interfaces to specify bandwidth demands, while leave the estimation to applications themselves. Since existing techniques in traffic engineering provide good accuracy in predicting demand matrices, applications can know their long-term network profiles [5], [18] or calculate the demands on the fly [15].

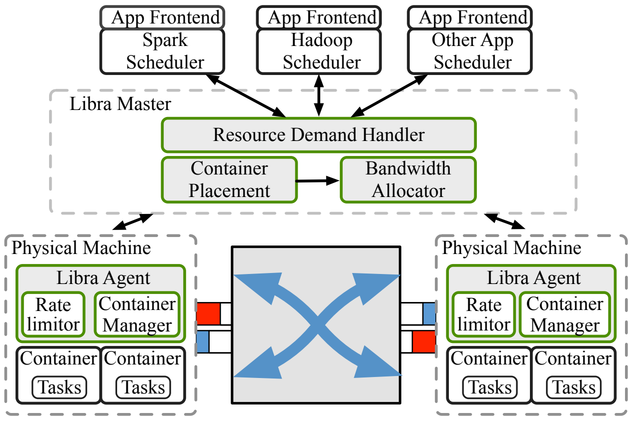


Fig. 3. The overview of *Libra*'s architecture.

Once received all the resource demands from concurrent applications, *Libra*'s resource demand handler first goes through an admission control procedure to validate security credentials of the applications and determine whether currently available resources can satisfy their computation demands. For admitted applications, the manager first runs the container placement module to determine which machine each container should be assigned to, and then runs the bandwidth allocator to compute the amount of bandwidth each container can use. After the containers are launched, the rate limiter on each machine manages the actual bandwidth consumption of each container. The conceptual architecture is shown in Fig. 3. *Libra* does not intervene task scheduling of applications. Each application can adopt its own task scheduler to decide how to use the allocated containers and resources.

Admission Control: Apart from security verification, the admission control also evaluates whether the hard constraints (such as the number of CPU cores and the amount of memory space) of the incoming applications can be satisfied. If not, only a subset of the incoming applications will be admitted based on the priorities and feasibility. The unqualified applications are withheld until more resources become available.

Fault tolerance: *Libra* circumvents the deficiency of a single manager leveraging ZooKeeper [19] as in existing centralized cluster management systems [1], [2], [3]. In case *Libra* completely fails, the default cluster manager will offer idle resources to applications in a max-min manner, and rely on the scheduler of each application to choose the right containers that satisfy their demands.

Scability: *Libra* recalculates applications' shares and redistribute bandwidth whenever any application arrives or departures. Fortunately, the time needed for recomputing and communicating the new allocations is largely offset by the consequent performance gains as demonstrated in large-scale simulations and testbed-based experiments in Sec. V-B1.

B. Handling Dynamics

The network sharing algorithms proposed in Sec. III can successfully improve isolation guarantee without sacrificing network utilization in offline cases. However, applications may come and leave dynamically in production clusters. We present how *Libra* deals with such situations in Alg. 1.

When an existing application departs, its containers will be terminated and the corresponding resources will be released. Since it is possible that the system bottleneck with all the active applications changes consequently, we detect the new bottleneck and re-calculate the isolation guarantee for each application accordingly (line 15-17). We involve a dynamic filling step in *Libra* to enable containers of active applications consume the idle local bandwidth if necessary.

When a new application registers with the cluster manager, *Libra* tries to satisfy its resource demands without migrating existing containers. We first restrict the bandwidth usages of running containers to their minimum guaranteed bandwidth. We then use the spared bandwidth as resource constraints to place the containers of the incoming application (line 11-14 of Alg. 1).

Algorithm 1 The Libra Framework

```
 1: procedure DEMANDHANDLER($\{A_i\}$)
 2:     Initiate the demand vector $\{v_{ij}^k\}$ for each pending application $A_i$
 3:     Monitor the available resources of each machine $M^k$
 4:     Sort machines based on their remaining bandwidth
 5:     The top $\kappa$ machines form the candidate set $\{M_{\text{candidate}}\}$
 6:     $\{x_{ij}^k\}$=MINIMUMBOTTLENECK($\{A_i\}, \{M_{\text{candidate}}\}$)
                                    ▷ Determine where to place each container
 7:     MAXIMUMISOLATION($\{x_{ij}^k\}$)
                                    ▷ Determine the bandwidth limit of each container
 8: procedure MAIN
 9:     Initiate the pending applications in the cluster as $\mathcal{A}$
10:     DEMANDHANDLER($\mathcal{A}$)
11:     if an application $A_n$ arrives then
12:         Restrict the bandwidth limits to
13:         $\alpha_{ij}^k = u_{ij}^k/\mathcal{B}$, $\beta_{ij}^k = d_{ij}^k/\mathcal{B}$
14:         DEMANDHANDLER($A_n$)
15:     if an application $A_o$ leaves then
16:         set $x_{oj}^k = 0$ for all its containers
17:         MAXIMUMISOLATION($\{x_{ij}^k\}$)
18:     if an application $A_i$ changes demands then
19:         MAXIMUMISOLATION($\{x_{ij}^k\}$)
```

When an active application changes its network demands, we re-apply Alg. 3 to compute the new isolation guarantee and bandwidth restrictions. If an active application tries to terminate some of its containers, the corresponding resources will be released and re-allocated as in line 15-17. If an active application applies for new containers, this set of requests will be viewed as a new application and treated as in line 11-14. It may further reduce system bottleneck and thus improve isolation guarantee if we allow live container migration whenever application arrives in or leaves the system. However, live migrations will incur extra network traffic due to the footprint of active flows and the update of forwarding rules [20] [21]. Since we wish to minimize the influence on running applications, we do not enable container migration for now, and allow the applications themselves to periodically migrate their containers using existing schemes [22], [23].

C. Network Sharing Algorithms

Given the demands of applications and the available resources of machines in the cluster, we first select a subset of candidate machines based on their remaining bandwidth. We are then ready to solve the generalized bottleneck problem stated in Eq. (4) using an LP solver. After retrieving the fractional values for x_{ij}^k, we use the rounding technique to set $x_{ij}^{k_0} = 1$ where $k_0 = \arg\max x_{ij}^k$ (line 4-7 of Alg. 2). A container c_{ij} is then placed onto the machine M^k where $x_{ij}^k = 1$.

Algorithm 2 Minimizing the System Bottleneck

```
1: Procedure MinimumBottleneck($\{A_i\}$,$\{M^k\}$)
2: Initiate: $(d_{ij}^k, u_{ij}^k, r_{ij}^k, c_{ij}^k)$ for each container
                              ▷ the machine-normalized demand vectors
3: Solve Problem (4) with the demands and machine states
4: for all container $c_{ij}$ in application $A_i$ do
5:     Initialize $x_{ij}^k = 0$ for all $k$
6:     $k^* \leftarrow \arg\max_k x_{ij}^k$
7:     $x_{ij}^{k^*} = 1$
8: Adopt Tabu search and update $\{x_{ij}^k\}$
9: Deploy a container $c_{ij}$ on the machine $M^k$ where $x_{ij}^k = 1$
```

Once we determined where to place each container of the pending applications, we can compute the demand for network bandwidth on each physical link and thus locate the system bottleneck whose normalized demand is the largest (line 3-5 of Alg. 3).

Libra handles the demand conflicts on the bottleneck link based on dominant resource fairness [9]. For other network links, *Libra* first allocates bandwidth in a way that containers of the same application achieve the same isolation guarantee. *Libra* then applies a backfilling step to allow containers on the same machine share the unallocated bandwidth in proportion to their demands. We further allow a container to temporarily reclaim the idle bandwidth allocated to other containers and immediately gives up such extra allocation once the original container calls for more bandwidth, or new requests from incoming applications arrive. The allocation results of Fig. 1(b) under different strategies are shown in Fig. 4.

We can see that *Libra* not only results in a better placement situation where all the allocation strategies can achieve better isolation, but also increases the link utilization through backfilling. But as analyzed in HUG [5], such higher link utilization comes at the expense of losing strategy-proofness. We only apply our backfilling strategy in private clouds where applications do not lie about their demands on purpose, and leverage HUG's backfilling restrictions to impose strategy-proofness in public clouds.

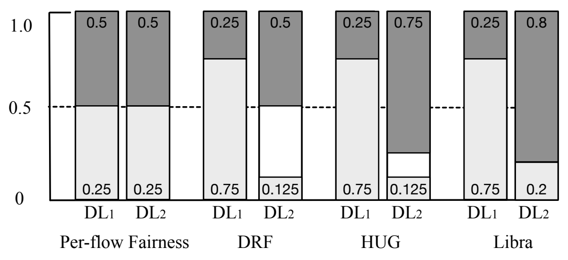


Fig. 4. Revist the example in Fig. 1(b): different bandwidth allocation strategies result in different network utilization: DRF leaves 0.375 Gbps of DL_2 idle; HUG improves utilization but still leaves 0.125 Gbps bandwidth unallocated. *Libra* always allocates all the bandwidth by default.

Algorithm 3 Maximizing the Isolation Guarantee

```
1: Procedure MaximumIsolation($\{x_{ij}^k\}$)
2: Initialize the bottleneck demand $\mathcal{B}^* = 0$
3: for all $M^k$ do
4:     if $U^k > \mathcal{B}^*$ then $\mathcal{B}^* = U^k$
5:     if $D^k > \mathcal{B}^*$ then $\mathcal{B}^* = D^k$
                              ▷ Find the value of the system bottleneck
6: for all $x_{ij}^k = 1$ do     ▷ proportionally allocate bandwidth
7:     if $U^k > 0$ then $\alpha_{ij}^k = \mathcal{B}^* u_{ij}^k$
8:     if $D^k > 0$ then $\beta_{ij}^k = \mathcal{B}^* d_{ij}^k$
9: for all machine $M^k$ do
10:    Backfill the unallocated bandwidth to containers
```

V. Experimental Evaluation

We evaluated *Libra* through our testbed deployment on the Linode cloud hosting platform [24] as well as via large-scale simulations.

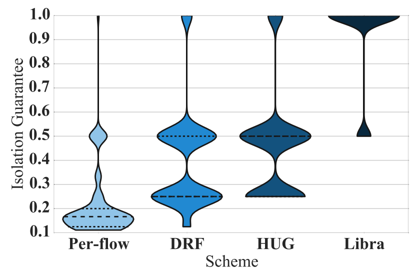
Schemes to compare: We compare the following schemes with *Libra*.

- Baseline: all the containers are placed onto machines in a round-robin fashion without considering network demands as in YARN [2], and all the flows follow per-flow fairness without any proactive rate limiting.
- Isolation-oriented (DRF): round-robin container placement with flows following dominant resource fairness [9] to ensure application isolation.
- Isolation with high utilization (HUG): round-robin container placement with flows rate-controlled based on the policy proposed by HUG [5].

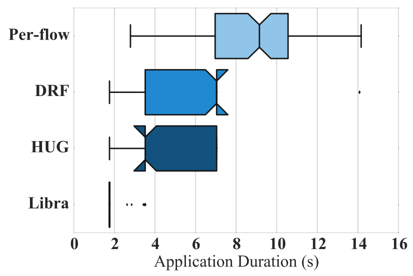
Through comparisons with the related schemes, we can inspect the benefits brought by the two ingredients of *Libra*: isolation-aware container placement and bandwidth allocation.

Performance Metrics: To comprehensively compare *Libra* with existing schemes, we examine three performance metrics, isolation guarantee, application performance and link utilization. We define the performance gain of scheme 1 to scheme 2 as $\frac{|\mathrm{metric}_1 - \mathrm{metric}_2|}{\mathrm{metric}_2}$, where metric_1 and metric_2 are the metric value derived by scheme 1 and scheme 2, respectively.

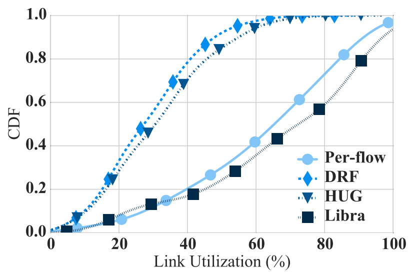
Summary of the **main results** is as follows:



(a) Isolation Guarantee



(b) Application Duration



(c) Link Utilization

Fig. 5. Testbed experiments: a) in comparison with per-flow fairness, DRF and HUG on top of YARN, *Libra* improves isolation guarantee by 327.89%, 132.18% and 105.59% respectively; b) *Libra* speeds up applications by 77.18%, 63.94% and 57.71% respectively; c) with *Libra*, the utilizations of about 65.9% links are higher than 60% (links that were never used during the experiments are excluded).

- *Libra* improves the optimal isolation guarantee through isolation-aware container placement: the optimal isolation is 3.27× more than the baseline (per-flow fairness on top of YARN) and about 1.32× and 1.05× more than DRF and HUG in testbed experiments (Sec. V-A). Its benefits are relatively stable under different network scenarios (Sec. V-B2 and V-B3).
- Due to improved isolation, *Libra* improves link utilization and application performance as well: the performance gain in terms of application durations is about 75.51%, 70% and 68.05% than the baseline, DRF and HUG (Sec. V-B).
- *Libra* isolates multiple concurrent applications across the entire network, and it can scale up to 30,000 machines with less than 1 second overhead. (Sec. V-B1).

A. Implementation and Testbed Experiments

We first microbenchmark *Libra* on a 30-node Linode cluster to evaluate its benefits. Each node is equipped with 6 CPU cores, 8 GB memory, 192 GB SSD storage, 1 Gbps uplink and 40 Gbps limit downlink, as predefined and fixed by Linode.

To encourage the public adoption of our cluster manager and to make the deployment of *Libra* smooth, we developed a set of plugins around the popular DevOps tool, Ansible [25], to automatically configure and launch up all the nodes.

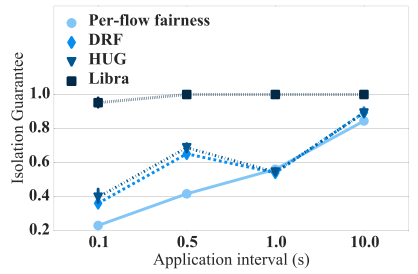
1) Implementation Details: The cluster manager runs on top of the master node, which accepts requests from application clients, and manages all resources of the slave nodes. An application first registers with the cluster manager to apply for the containers they desire. The manager then runs its placement and allocation strategies to determine on which machine that container should be placed and how many resources it should be allocated. To solve the problem in Eq. 4, we embed the API provided by CPLEX 12.6.3 [26] into our codebase. The communication among the master, slaves and application clients are all done in the format of `Protobuf` [27]. Slaves periodically reports their network and computation states to the master. Whenever necessary, multiple messages heading to the same machine are packed together. The delimitation of the messages are achieved by using the simple `Netstring` protocol [28]. These together ensures fast message exchange. We heavily use the `actor` model in our codebase to dispatch messages, such that we achieve high concurrent performance without worrying about race conditions.

The Linux kernel version of each node is 4.5.0, with all `cgroup` capabilities enabled to tag outgoing packets from different containers. We define `tc` filters based upon the `net_cls` tag of each container. The outgoing packets are then put into different `htb` classes of varying rate limits. Since the `net_cls` tag is only useful for shaping the outgoing packets, we police the incoming packets based upon the port range of the destinations. It turns out that the bottleneck mostly resides in the uplinks not the downlinks, as indicated by our supplementary tests on other popular cloud hosting platforms, *e.g.*, DigitalOcean and Vultr.

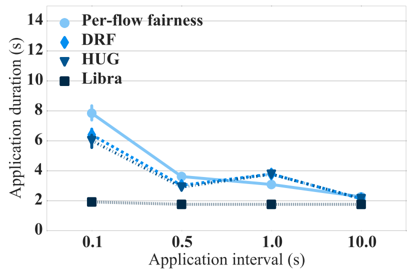
2) Characteristics: To better under the behavior of different schemes, we first take a scrutiny on one run of the schemes over the same arrivals of applications.

Applications arrive in the cluster with their intervals following an exponential distribution (average = 5s) [29]. 10 applications with their network demands in accordance with the characteristics observed in production clusters [5], [6] are submitted to the system. Each container requires for 1 CPU and 1 GB memory. Six applications have all-to-all communication pattern, while the other four applications have pairwise one-to-one communication. Without loss of generality, we assume all applications have the same weight.

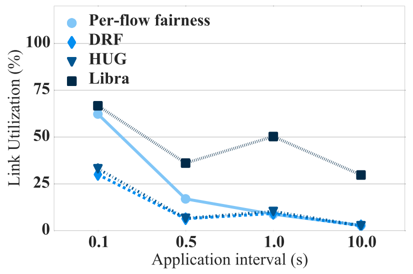
From Fig. 5(a) we can see that with our isolation-aware container placement, the demands of eight applications are completely satisfied. In contrast, with per-flow fairness and DRF on top of YARN, the isolation guarantees for most applications are less than 0.5. HUG presents better results despite it directly adopts the dominant resource fairness for maximizing isolation guarantee. The reason for such improvement stems from the better link utilization obtained by HUG as shown in Fig. 5(c). With better network utilization, the applications are able to finish faster, and thus there are more idle bandwidth for active applications to share. However, the improvements are relatively limited since HUG restricts backfilling to ensure strategy-proofness. *Libra* is able to significantly improves link utilization through work-conserving backfilling. Such benefit also comes from the more balanced distribution of network



(a) Isolation Guarantee

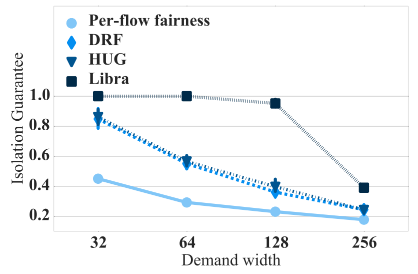


(b) Application Duration

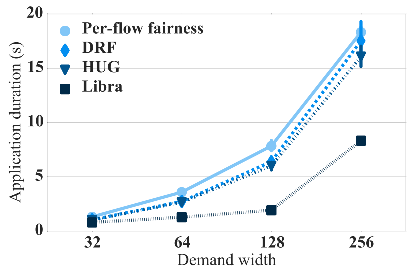


(c) Link Utilization

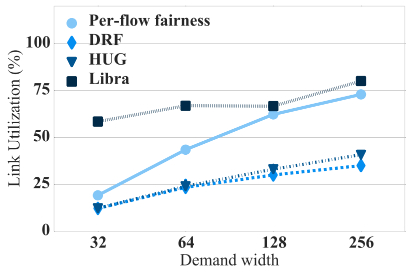
Fig. 6. Impact of inter-application arrival intervals: larger arrival intervals lead to less intense network competition. As a result, the isolation guarantee increases, while the application duration and link utilization decreases.



(a) Isolation Guarantee



(b) Application Duration



(c) Link Utilization

Fig. 7. Impact of application demands: with each application requiring more containers, the aggregated demand for network becomes more. The isolation guarantee thus decreases, while the application duration and link utilization increase at the same time.

demands that our container placement achieves. With better isolation guarantee and link utilization, *Libra* reduces the average application duration (Fig. 5(b)) with respect to per-flow fairness, DRF and HUG by $\frac{8.872-2.024}{8.872} = 77.18\%$, $\frac{5.614-2.024}{5.614} = 63.94\%$ and $\frac{4.787-2.024}{4.787} = 57.71\%$.

B. Large Scale Performance Analysis

Simulation Methodology: Similar to [5], [9], we developed our own event-driven simulator to imitate applications running in a data-parallel computing cluster. The simulator only accounts for the application arrival and departure events to reduce the simulation complexity. It updates the rate and remaining volume of each flow of an application when an event occurs.

In the simulations, we suppose applications have pairwise many-to-many communication pattern [5] and assume the application arrivals follow a Poisson distribution [29]. We first evaluate *Libra*'s overheads by inspecting the time consumed in computing the placements and allocations in Table. II.

We then evaluate three aspects that may affect the performance of *Libra* and other schemes: the inter-application arrival intervals, the width of applications (i.e., the number of containers an application requires), and the scope of the optimization procedure. For reasonable simulation time, we simulate 512 machines, each of which is connected to the network using 1 Gbps bisection bandwidth. In the simulations, each of our results is an average of 30 tries. The corresponding simulation results are shown in Fig. 6 and 7 and Table III.

1) ***Scalability and Overhead:*** The key challenge in scaling *Libra* is its centralized LP solver, which must recalculate application shares and redistribute the bandwidth across the entire cluster whenever any application arrives. The following table shows the computation time needed for different application widths and different percentage of machines as candidates.

TABLE II
LP COMPUTATION OVERHEAD (MILLISECONDS)

Percentage	Width=32	Width=64	Width=128
5.86%	3.08	5.06	5.64
10%	3.56	5.19	8.22
15%	6.43	9.57	16.83

We found that the time to calculate new placements using *Libra* is less than 3 microseconds in our 30 machine cluster. Furthermore, a re-computation due to an application's arrival or departure would take about 8.22 milliseconds on average for the 512 machine cluster. Even with larger clusters, the computation time will not grow exponentially since the size of our candidate set is determined by demand widths, not the cluster size. Communicating the placement and allocation decisions takes less than 5 milliseconds to 30 machines and around 0.5 second for 30, 000 emulated machines.

2) ***Impact of application dynamics:*** We vary the value of the average inter-application arrival time to examine the influence of dynamic incoming applications. In each round of simulations, 100 application with the same demand arrives in the network: each application requires for 128 containers, and each container tries to use up the available bandwidth (*i.e.*,

1Gbps). Note the larger arrival interval indicates the smaller arrival intensity. From the results in Fig. 6, we make the following observations.

Firstly, as shown in Fig. 6(a), the isolation guarantee increases with the increase of the average inter-application arrival interval. The reason for this is that a larger interval indicates less concurrent applications in the network. The contention for network resources is thus not that intense. Furthermore, we find that *Libra* is less sensitive to the variation of arrival intervals. Even with 10 applications arrive every second, it is able to ensure over 90% of their demands can be satisfied. Compared to per-flow fairness, DRF and HUG, the performance gains are as large as $\frac{0.952-0.231}{0.231} = 312.12\%$, $\frac{0.952-0.361}{0.361} = 163.71\%$ and $\frac{0.952-0.397}{0.397} = 139.80\%$ (when arrival interval is 0.1s).

Secondly, with higher isolation guarantees, applications get allocated more bandwidth to transfer their data. Therefore, *Libra* is able to significantly reduce application durations as shown in Fig. 6(a). Compared to the other schemes, the performance gains are as large as $\frac{7.84-1.92}{7.84} = 75.51\%$, $\frac{6.40-1.92}{6.40} = 70\%$ and $\frac{6.01-1.92}{6.01} = 68.05\%$. With larger arrival intervals, DRF and HUG catch up with *Libra* gradually. Finally, the average link utilization decreases with the increase of inter-application arrival intervals since less applications usually imply less demand for link bandwidth. The performance gain of *Libra* ranges between 105.9% (interval=0.1s) to 1031.1% (interval=10s).

*3) **Impact of application demands**:* To evaluate how the performance is influenced by the application demands in the network, we fix the arrival intervals to 0.1 second. As the amount of link bandwidth each container requires is the same, the only degree of freedom is the number of containers each application requires (denoted as the width of its demand). We vary the average demand width between 32 to 256 in our experiments since the statistics reported in the production trace [5] indicates that about 68% of applications require less than 50 containers. From the results in Fig. 7, we make the following observations.

The trend of the curves in Fig. 7(a) shows that the isolation guarantee achieved by all the schemes decrease with the increase of demand widths. This is obvious since the concurrent demands for bandwidth essentially increase with a larger demand width. When each application only asks for 32 containers, the performance gain of *Libra* with respect to DRF and HUG is only $\frac{1-0.848}{0.848} = 17.88\%$ and $\frac{1-0.8615}{0.8615} = 16.04\%$. However, *Libra* is able to maintain high isolation guarantee when the average width increases to 128: the isolation guarantee only slightly drops from 1.0 to 0.952. Without network-aware container placement, the isolation guarantees achieved by DRF and HUG drop to 0.361 and 0.397 respectively. When the demand width continues to increase, the benefit of *Libra* becomes less significant with the performance gain decreasing to 60.76% and 59.82%. As stated above, larger demand widths lead to more bandwidth demand and more fierce network contention. As a result, the application durations and link utilizations achieved by all the schemes increase accordingly (Fig. 7(b) and Fig. 7(c)).

TABLE III
NORMALIZED APPROXIMATION RATIO (ISOLATION GUARANTEE)

Percentage	Width=32	Width=64	Width=128
5.86%	0.98	0.92	0.90
10%	0.99	0.95	0.92
15%	0.99	0.97	0.97

*4) **Impact of Optimization Scope**:* From the table above we can see that although we only choose the machines with the most available bandwidth as candidates, the performance degradation is rather small. In our 512-node cluster, choosing from 5.85% of nodes (namely, 30 candidates) can result in 0.9-approximate solutions, which means the resultant isolation guarantee is within 10% of the global optimal isolation. This demonstrates that reducing the selection scope has little side effects given the largely improved efficiency.

VI. RELATED WORK

Cluster management: The offer-based cluster managers, such as Mesos [1] and Borg [3], dynamically allocates idle resources (as the form of containers) to applications based on different policies. Such departures from traditional static partitioning of resources significantly improves cluster utilization. The resource manager in YARN [2] acts as the central authority arbitrating resources requests from competing applications in the cluster. By allowing applications to specify their location preferences and the properties of the desired containers, YARN enforces rich policies for global control. However, similar to offer-based schemes, YARN only captures computation and storage resources, and still lacks network-awareness. *Libra* further improves existing cluster managers by considering the demand for network resources when allocating containers, and make applications efficiently share the cluster with network isolation guarantee.

VM placement and migration: Most existing placement schemes either restrict themselves to computation and storage resources or only focus on minimizing the aggregated cross-rack traffic traversing the core network [22], [30], [31]. Ignoring the demand and supply relationship on each access link, it is possible that some links become hot spots, and harms the isolation guarantee of relevant applications. Cohen *et al.* [23] propose to maximize the aggregated available bandwidth each VM can get. Although this scheme is bandwidth-aware, it provides no performance guarantee and neglect application semantics. Alicherry *et al.* in [32] focus on how to minimize data access time when placing virtual machines. All the schemes mentioned above cannot provide network isolation guarantee either at the system or the application level.

Bandwidth allocation: Strategies [33], [14] that rely on reservation to achieve bandwidth isolation fail to achieve high network utilization. Depending on the level of isolation, existing work in dynamic bandwidth sharing can be divided into flow-level [34], VM-level and network-level schemes. Both flow-level and VM-level schemes [35] provide no performance guarantee. DRF achieves optimal isolation guarantee and strategy proofness at the cost of low utilization [9]. PS-P [6] and EyeQ [7] try to provide bandwidth guarantee through work-conserving rate limiting but only achieves suboptimal performance guarantee. HUG [5] further optimizes the isolation guarantee with improved link utilization. All these schemes try to achieve bandwidth guarantee given the placement of containers. In contrast, we perform network-aware container placement to minimize the system bottleneck and thus improve the optimal isolation guarantee that any bandwidth allocation scheme can achieve.

VII. Conclusion

In this paper, we have proposed, *Libra*, a cluster management framework that improves network isolation guarantee without the loss of network efficiency. To the best of our knowledge, *Libra* is the first work that proposes and leverages the fact that container placement determines the optimal network isolation that can be achieved by any bandwidth allocation mechanism. However, the binary nature of container placement makes the problem almost intractable. Despite such hardness, we propose a two-stage algorithm that effectively place containers and allocate bandwidth. First we place containers to physical machines to minimize the system bottleneck; then we allocate bandwidth among co-located containers of different applications to achieve the optimal isolation guarantee. Through both testbed experiments and extensive simulations, we demonstrate that *Libra* works with existing cluster managers and preserves remarkable performance advantages over existing network-unaware placement and allocation-only solutions. *Libra* is also easy to deploy with our customized configuration tools, such that existing cluster management frameworks can take advantage of *Libra*.

References

[1] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, “Mesos: a platform for fine-grained resource sharing in the data center,” in *Proc. USENIX NSDI*, 2011.

[2] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache Hadoop YARN: Yet Another Resource Negotiator,” in *Proc. Symposium on Cloud Computing*, 2013.

[3] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, 2015, pp. 18:1–18:17.

[4] “Docker: an open platform for distributed applications for developers and sysadmins,” https://www.docker.com/.

[5] M. Chowdhury, Z. Liu, and I. Stoica, “HUG: Multi-resource fairness for correlated and elastic demands.” NSDI, 2016.

[6] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, “FairCloud: sharing the network in cloud computing,” in *Proc ACM SIGCOMM*, 2012.

[7] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg, “EyeQ: Practical network performance isolation at the edge,” in *Proc. USENIX NSDI*, 2013.

[8] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” in *Proc. USENIX OSDI*, 2004.

[9] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: Fair allocation of multiple resource types,” in *Proc. USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011, pp. 323–336.

[10] S. Martello and P. Toth, “The bottleneck generalized assignment problem,” *European journal of operational research*, vol. 83, no. 3, pp. 621–638, 1995.

[11] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, 2008.

[12] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, “BCube: a high performance, server-centric network architecture for modular data centers,” *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009.

[13] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks,” in *Proc. USENIX NSDI*, 2010.

[14] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, “SecondNet: a data center network virtualization architecture with bandwidth guarantees,” in *Proc. ACM CoNEXT*, 2010.

[15] M. Chowdhury, Y. Zhong, and I. Stoica, “Efficient Coflow Scheduling with Varys,” in *Proc. ACM SIGCOMM*, 2014, pp. 443–454.

[16] Ö. Karsu and M. Azizoğlu, “The multi-resource agent bottleneck generalised assignment problem,” *International Journal of Production Research*, vol. 50, no. 2, pp. 309–324, 2012.

[17] F. Glover, “Tabu searchpart i,” *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[18] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma, “Application-driven bandwidth guarantees in datacenters,” in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 467–478.

[19] “Apache ZooKeeper,” https://zookeeper.apache.org/.

[20] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proc. ACM SIGCOMM conference on Internet measurement (IMC)*, 2010.

[21] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, “The nature of data center traffic: measurements & analysis,” in *Proc. ACM SIGCOMM conference on Internet measurement conference (IMC)*, 2009, pp. 202–208.

[22] F. P. Tso, K. Oikonomou, E. Kavvadia, and D. P. Pezaros, “Scalable traffic-aware virtual machine management for cloud data centers,” in *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2014, pp. 238–247.

[23] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, “Almost optimal virtual machine placement for traffic intense data centers,” in *Proc. IEEE INFOCOM*, 2013, pp. 355–359.

[24] “Linode: SSD Cloud Hosting,” http://www.linode.com/.

[25] “Ansible is simple IT automation,” http://www.ansible.com/.

[26] “IBM ILOG CPLEX Optimization Studio V12.6.3,” http://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.3/ilog.odms.studio.help/Optimization_Studio/topics/COS_home.html.

[27] “Protocol Buffers - Google’s data interchange format,” https://developers.google.com/protocol-buffers/.

[28] D. J. Bernstein, “Netstrings,” https://cr.yp.to/proto/netstrings.txt.

[29] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling,” in *Proc. European conference on Computer systems (Eurosys)*, 2010, pp. 265–278.

[30] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma, “Application-driven bandwidth guarantees in datacenters,” in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 467–478.

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

[32] M. Alicherry and T. V. Lakshman, “Optimizing data access latencies in cloud systems by intelligent virtual machine placement,” in *Proc. IEEE INFOCOM*, 2013, pp. 647–655.

[33] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, “Silo: Predictable message latency in the cloud,” in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, 2015, pp. 435–448.

[34] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center TCP (DCTCP),” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 63–74, 2011.

[35] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, “Sharing the data center network,” in *Proc. USENIX NSDI*, 2011.