Facilitating Application-Aware Bandwidth Allocation in the Cloud with One-Step-Ahead Traffic Information

Dian Shen, Junzhou Luo, *Member, IEEE*, Fang Dong, Jiahui Jin, Junxue Zhang, and Jun Shen, *Senior Member, IEEE*

Abstract—Bandwidth allocation to virtual machines (VMs) has a significant impact on the performance of communication-intensive big data applications hosted in VMs. It is crucial to accurately determine how much bandwidth to be reserved for VMs and when to adjust it. Past approaches typically resort to predicting the long-term network demands of applications for bandwidth allocation. However, lacking of prediction accuracy, these methods lead to the unpredictable application performance. Recently, it is conceded that the network demands of applications can only be accurately derived right before each of their execution phases. Hence, it is challenging to timely allocate the bandwidth to VMs with limited information. In this paper, we design and implement *AppBag*, an *App*lication-aware *Ba*ndwidth *g*uarantee framework, which allocates the accurate bandwidth to VMs with one-step-ahead traffic information. We propose an algorithm to allocate the bandwidth to VMs and map them onto feasible hosts. To reduce the overhead when adjusting the allocation, an efficient Lazy Migration (LM) algorithm is proposed with bounded performance. We conduct extensive evaluations using real-world applications, showing that *AppBag* can handle the bandwidth requests at run-time, while reducing the execution time of applications by 47.3 percent and the global traffic by 36.7 percent, compared to the state-of-the-art methods.

Index Terms—Bandwidth allocation, virtual machine, application-aware, cloud computing, data center network

1 Introduction

Modern virtualization based cloud data centers, such as Amazon EC2 [1], are becoming the hosting platform for a wide spectrum of big data applications, including online data mining [2], [3], social network analysis [4], scientific computing [5], [6] and etc. Many of these emerging big data applications introduce intensive network traffic among the hosting virtual machines (VMs). Sharing the network infrastructure, VMs contend for the scarce network resources with co-resident VMs, leading to unpredictable performance of the encapsulated applications [7]. This raises concern with respect to how to guarantee the bandwidth of VMs in the cloud data centers.

[1]Tackling the temporal and spatial variability of applications communication traffic, bandwidth guarantee for VMs is achieved via dynamic bandwidth reservation [7], [9], [10], [11], [12], [13], [14]. Prior approaches typically provide bandwidth guarantees based on prediction techniques. They predict the network demands of applications for a long term either through time series analysis [9], [15] or execution sampling [10], [16]. However, these methods are deficient in prediction accuracy and flexibility, resulting in over-allocating or insufficient bandwidth for VMs, hence the unpredictable performance. From our experimental analysis in Section 2.3, the execution time of applications could vary from $1.2\times$ to $1.9\times$ due to the erroneous bandwidth allocation.

The difficulty of accurately predicting VMs' network demands stems from the unawareness of applications' execution information. As is depicted in Fig. 1, the traffic fluctuations actually coincide with the interleaving of applications' execution phases and can be predicted nearly 100 percent right before each execution phase starts [17]. Given the accurate network demands, the cloud providers can allocate the bandwidth to each VM precisely, benefitting both the customers and the providers with more predictable application performance and improved utilization of infrastructure, respectively.

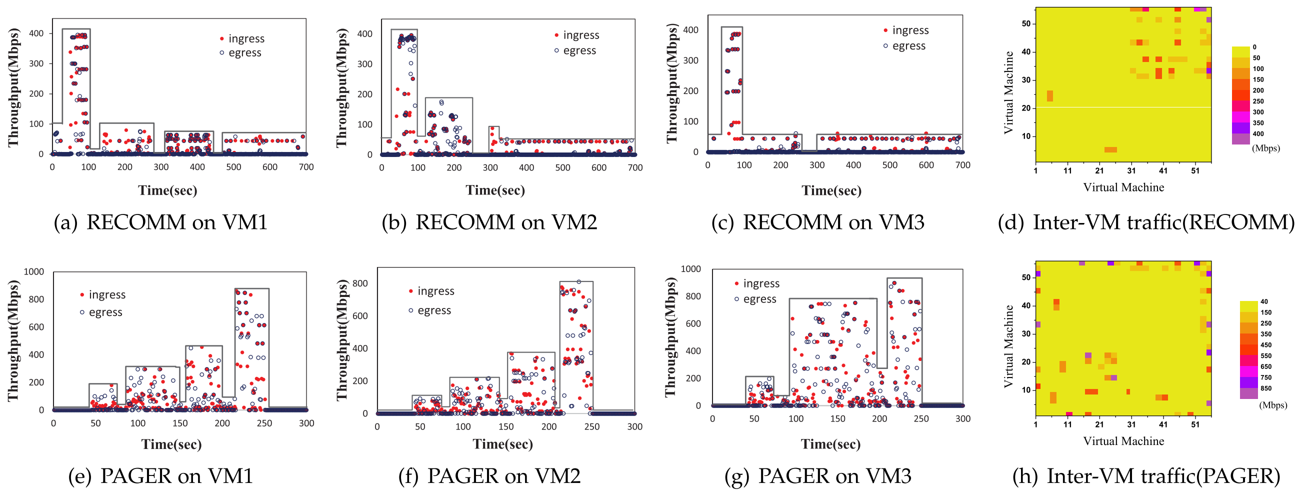
Therefore, it is of significant importance to design a bandwidth allocation framework for cloud data centers that:

- *Offers the accurate bandwidth guarantees*: Each VM is guaranteed a minimum absolute bandwidth according to the precise demands of applications, subject to the physical constraints.

1. *Preliminary version [8] appeared in Proceedings of the 45th International Conference on Parallel Processing (ICPP 2016) as a regular research paper.*

- *D. Shen, J. Luo, F. Dong, and J. Jin are with the School of Computer Science and Engineering, Southeast University, Nanjing 210018, China. E-mail: {dianshen, jluo, fdong, jhjin}@seu.edu.cn.*
- *J. Zhang is with SING Group, Hong Kong University of Science and Technology, Hong Kong, China. E-mail: jzhangcs@ust.hk.*
- *J. Shen is with the School of Computing and Information Technology, University of Wollongong, NSW 2522, Australia. E-mail: jshen@uow.edu.au.*

Manuscript received 15 June 2017; revised 27 Dec. 2018; accepted 5 June 2019. Date of publication 11 June 2019; date of current version 15 Apr. 2020. (Corresponding author: Fang Dong). Digital Object Identifier no. 10.1109/TSC.2019.2922176



(a) RECOMM on VM1 (b) RECOMM on VM2 (c) RECOMM on VM3 (d) Inter-VM traffic(RECOMM)

(e) PAGER on VM1 (f) PAGER on VM2 (g) PAGER on VM3 (h) Inter-VM traffic(PAGER)

Fig. 1. Traffic patterns of cloud applications.

- *Uses only one-step-ahead information*: The bandwidth is allocated for the next execution phase of hosted applications. It is dynamically adjustable before their future execution phase starts.
- *Is practical*: The framework should be fully compatible with and readily deployable in the current cloud data centers. With the complexity of this kind of problem, the overhead of the algorithm, virtualization and implementation should be considered and reduced.

Existing research for sharing cloud networks, e.g., [9], [10], [11], [12], [13], [14], could not achieve all of the above goals simultaneously. Our earlier work [8] took the first step towards achieving all of the above properties. We had presented the feasibility study in allocating bandwidth using only limited information with *AppBag*. The rationale of the method is to derive an approximately optimal solution based on the one-step-ahead information and then extend the solution to subsequent phases. However, our previous approach still faces two challenges: (i) due to the lack of theoretical analysis and comprehensive experiments, the applicability of application-aware bandwidth allocation to more complicated real-world scenario remains uncertain. (ii) as it is a tradeoff between traffic-aware VM placement and bandwidth adjustment, to strike a balance between these two functionalities is still challenging.

In this paper, to tackle the above limitations, we move further to complement *AppBag* in the following aspects.

First, as the heuristic-based approach of bandwidth adjustment suffers from unbounded performance in practice, we revamp an online algorithm, Lazy Migration (LM), which near-optimally solves the problem of online bandwidth allocation. By means of in-depth theoretical analysis, we prove that LM algorithm, with only one-step-ahead information, is $\mathcal{O}(\frac{1}{c} + d)$-competitive. (Section 4.3)

Second, the impact of the key parameter β that bridges the two key ingredients of *AppBag* is comprehensively investigated. We demystify its impact with a measurement study and quantitatively analyze the detailed procedure of VM migration, so that we can offer the administrators the guidance of setting this key parameter in practice. (Section 4.1)

Third, we extend the system design and implementation of *AppBag* by leveraging the flexibility and high programmability of software-defined network edge. Through pushing the functionality of one-step-ahead bandwidth allocation in the kernel on the hypervisor, we implement this module as an extension of current Linux kernel module, supporting run-time plug-in, with no modification in transport layer, customer VMs, or networking hardware. (Section 3)

Fourth, with all the theoretical and system improvement, our method can be adapted to more complicated real-world scenarios, in particular cases of significant traffic variation and heavy application arrival rate. We demonstrate the enhanced capability of *AppBag* through 3 new test cases and 9 comparisons. To highlight a few, herein under heavy arrival rate, *AppBag* can maintain a rejection rate as low as 11.2 percent. The measured system occupation and network occupation rates are 24 and 17.5 percent lower, respectively. (Section 5.2.2)

The main contributions of this paper include that:

i) We explore the problem and design space of accurately allocating the bandwidth to VMs with only one-step-ahead prediction, which is a real-world constraint.
ii) Algorithms for bandwidth allocation and VM migration are proposed, with bounded performance and limited extra overhead.
iii) A prototype system *AppBag* is implemented based on OpenStack. The system is readily deployable in current data centers without hardware modification. We have deployed *AppBag* on an 18-server testbed in SEUCloud data center with more than 200 VMs.
iv) Extensive evaluations using several real-world applications have shown that our method can reduce the execution time of applications by up to 47.3 percent and the global traffic by up to 36.7 percent, compared to the state-of-the-art methods.

The rest of this paper is organized as follows. Section 2 describes the background and motivation of application-aware bandwidth allocation. Section 3 demonstrates the design and implementation of key components in the system. In Section 4, the problem is formulated and key algorithms are proposed. In Section 5, we present the evaluation results of this system. We discuss the related work and future work in Section 6 and conclude the paper in Section 7.

2 Background and Motivation

2.1 Traffic Patterns of Cloud Applications

To explore the traffic patterns of cloud applications, we conduct a measurement study. Our testbed consists of 18 physical hosts, each of which is equipped with two 6-core Xeon 2.66 GHz processors, 24 GB memory and a dedicated hard drive. For the benchmarking applications, we implement two representative applications, called RECOMM and PAGER. RECOMM is a Hadoop [20] based recommendation application, mining on the synthetic MovieLens [21] dataset. The PAGER application is based on a Pregel-like graph processing framework, implemented as the BSP [22] model, to compute page rank values over a synthetic web page dataset. Fig. 1 plots the traffic patterns of several representative VMs in a finite time horizon.

We observe the *temporal variation* that both applications experience dramatic traffic fluctuations during execution, which are in according with their execution phases. RECOMM consists of several MapReduce iterations, and the shuffle phases of each iteration are most communication-intensive. For PAGER, the execution consists of several BSP steps to compute the values. With more graph vertexes involved, the communication in each step mounts higher. The reality is, applications are able to know “which and where the data should be sent to” only after the former processing finishes. Therefore, their network demands can be predicted nearly 100 percent before the next execution phase starts [17]. It is also labor-saving if applications could specify their demands explicitly through predefined APIs.

We also see the *spatial variation* that the traffic distributions of both applications are highly uneven, depicted in Fig. 1d and 1h. For RECOMM, if the data is located at the same node where the task is scheduled, no data shuffle and transfer takes place. For PAGER, as the out-degree and in-degree of vertexes in the graph are different, some hot vertexes could be accessed more frequently in the computation. The skewed data distribution and popularity both lead to the uneven traffic distribution. This indicates a great potential in optimizing VM placement to reduce bandwidth usage by co-locating the chatty VM pairs in the same physical host.

2.2 Bandwidth Allocation Model

Pipe versus Hose and its Variations. Most of the previous research are based on the variants of the hose [12] model, where all VMs are connected to a central switch by a dedicated link (hose) with a minimum bandwidth guarantee B. Some typical variants include Virtual Oversubscribed Cluster (VOC) model [23], Tenant Application Graph (TAG) [24] and etc. All of these models have their advantages in representing certain applications. For example, hose model is especially useful in representing batch applications, and TAG captures the communication structure of layered applications very well.

However, they also have several limitations. First, they require the communication structure to be known as a priori or static during execution. Second, all of these models still make great efforts to convert the logical abstraction to the configuration of rate-limiters. In contrast, the VM-to-VM pipe model is general to most applications. For the administrators, the VM-to-VM pipe model is straight-forward to be converted to rate-limiters. Moreover, it suits well for the flow level network management(e.g., OpenFlow) and VM placement strategies. For the applications, the VM-to-VM pipe model is easy to derive, as the sender-to-receiver programming model can be converted straight-forward to the pipe model. Therefore, we choose VM-to-VM pipe model in this paper as it suits best in the application-aware VM bandwidth allocation context, agreeing with several related works [9], [14].

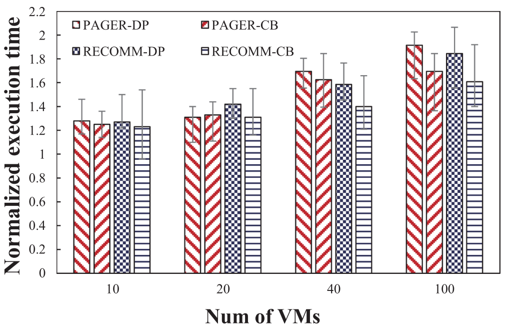


Fig. 2. The impact of prediction error.

Nevertheless, the major disadvantages of the VM-to-VM pipe model are computation intensiveness and the scalability issue. Hence, a series of measures have been taken in this paper to reduce the overhead. First, we consider only the large flows, leaving the small ones to legacy work-conserving methods, since the sizes of flows in the data center usually follow a long-tailed distribution that a few large flows consume most of the network resources [25]. Further, we decompose the global traffic matrix into individual ones for each application since normally there is very few traffic communication between applications. As the scale of traffic matrix for each application is limited, the algorithm overhead is significantly reduced, enabling our method to make online bandwidth allocation and VM placement decisions at run-time.

2.3 Traffic Forecasting

One-Step-Ahead Prediction versus Long Term Prediction. Conventionally, traffic forecasting methods provide long-term prediction based on the time series analysis [9] or execution sampling [10], [16]. However, the precision of such methods is not satisfying, especially when facing the complicated traffic patterns. Cicada [9] states that their method has a 5 -10 percent error rate. The latest work CODA [26] states that it identifies flows and their dependencies with around 90 percent accuracy.

In our experiment settings, we manually introduce random errors from 5 to 10 percent in terms of flow size, and evaluate its impact on the execution time of applications. We compare *AppBag* with other popular algorithms including the dynamic programming (DP) based method [10], [12] and the correlation based (CB) method [15]. Fig. 2 illustrates the normalized execution time of the two benchmarking applications PAGER and RECOMM under DP and CB. The bar shows the average results and the whiskers extend to show the 90th percentile results and the 10th percentile results. The average execution time is as much as $1.8\times$ and $2.1\times$ elongated for the two applications under DP. Although CB is less sensitive to the erroneous prediction, it still

TABLE 1
Summary of Previous Approaches and Comparison to *AppBag*

	Design decisions			System requirements		
Related work	BW guarantee model	Prediction model	Rate limiting	Switch hardware	Topology	Control model
Oktopus [12], SecondNet [13], CloudMirror [11]	Hose/VOC/TAG	N/A	End host/ Switch	Yes	No	Centralized
Proteus [10]	Hose	Long term prediction	Switch	No	Yes	Centralized
ElasticSwitch [14]	Hose/ Pipe	N/A	Hypervisor	No	No	Distributed
Cicada [9]	Pipe	Long term prediction	Hypervisor	No	No	Centralized
NumFabric [18]	Pipe	N/A	Switch	Yes	No	Distributed
HUG [19]	Hose	N/A	End host	No	No	Centralized
AppBag	**Pipe (Optimized)**	**One-step-ahead**	**Hypervisor (with SDN)**	**No**	**No**	**Centralized**

endures up to $1.2\times$ and $1.4\times$ elongation for RECOMM and PAGER, respectively.

Recently, HadoopWatch [17] has proposed the application layer traffic forecasting, which extracts the traffic demand information existing in the log and meta-data files of many big data applications. The prediction accuracy is nearly 100 percent. It is in consonance with our observation results that the fluctuation of traffic coincides with the interleaving of applications' execution phases. However, the accuracy of prediction comes at the price that it can only predict the traffic demands in the next one phase. In essence, the fact is true that only after the former processing finishes, the applications are able to know "which and where the data should be sent to". Hadoop [20], as an example, logs these information before mapping, shuffling and reducing phase. More generally, we provide a set of APIs for applications to register their network demands when available. Usually, these information are available not long before the transfer starts, which requires our method to determine the bandwidth allocation and instantiate it in a timely manner.

2.4 Rate Limiting

Hypervisor versus Switch versus End Host. Current technologies allow rate limiting on switches, end hosts(VMs) or hypervisors. On switches they do require the modification to hardware (e.g., supporting OpenFlow). Further, commodity switches usually have not enough resources to conduct VM-to-VM rate limiting, especially when there are large amount of VMs. Alternatively, many recent works resort to end host rate limiting. Linux Traffic Control and Iptables are proved to be very useful in this aspect. However, in the virtualized cloud environment, the end hosts are VM instances belonging to the tenants. The service provider should not modify the network components inside the tenants' VMs. Without control over tenants' VMs, the exertion of fine-grained bandwidth allocation remains at the mercy of VMs' networking components, which are usually non-cooperative. Therefore, to rate limit on the hypervisor layer is the most practical tradeoff. It is readily deployable by only installing the software on the hypervisors instead of modifications to the network topology and physical hardware. Furthermore, each hypervisor is only in charge of rate-limiting the VMs under its supervision. The flow table entries are very limited to the hosted VMs and ones they communicated with, introducing only some quite small overhead.

Overall, our method, subject to the real-world constraints, aims at providing a practical and readily deployable framework for data centers to provide accurate bandwidth allocation. Summary of previous approaches and comparison to *AppBag* is shown in Table 1.

3 Design and Implementation of *AppBag*

We implement *AppBag* based on OpenStack [27] and integrate it with SDN to manage VMs' network in the hypervisor layer. Inspired by [25], [28], we design *AppBag* to work in a centralized, cooperative manner. This is also coherent with many recent centralized designs for data center network [19], [29]. Overall, *AppBag* contains distributed agents on each VM to collect the real-time network demands, and a controller to quickly and efficiently determine the rate and scheduling upon the requests arriving. The architecture of *AppBag* is illustrated in Fig. 3. We functionally sketch out the implementations at the agents, the controller and the hypervisor.

Agents are pre-installed in the VM images. The agents support log-based one-step-ahead traffic prediction and also provide a set of APIs for the applications. The applications can invoke the APIs and interact with *AppBag* controller. The key APIs supported by *AppBag* are as follows.

`register`(*dst*, *rate*, [*option*]) is invoked to register a VM-to-VM bandwidth request with parameters to specify the destination IP address and the requested bandwidth. *[option]* here is used to force the request to overwrite the controller's policies, e.g., ignoring the small-rate requests, disabling migration. A result code is returned to indicate whether the request is instantiated or else, the reason of failure.

`unregister`(*dst*, [*option*]) is invoked to remove the bandwidth guarantee. A result code is returned to indicate whether the request is instantiated or else, the reason of failure. If the bandwidth guarantee is removed, the data transferred to specified destination should follow the legacy work-conserving manner, competing with other normal traffic, until `register`() is called.

`blocking`(*dst*, [*option*]) is invoked to suspend the data transfer to specified destination, until `register`() or `unregister`() is invoked.

The controller receives the bandwidth requests from APIs. Upon receiving requests, it first ignores the small ones whose rates are below a threshold. These small-rate flows are served as a work-conserving manner to fully utilize the

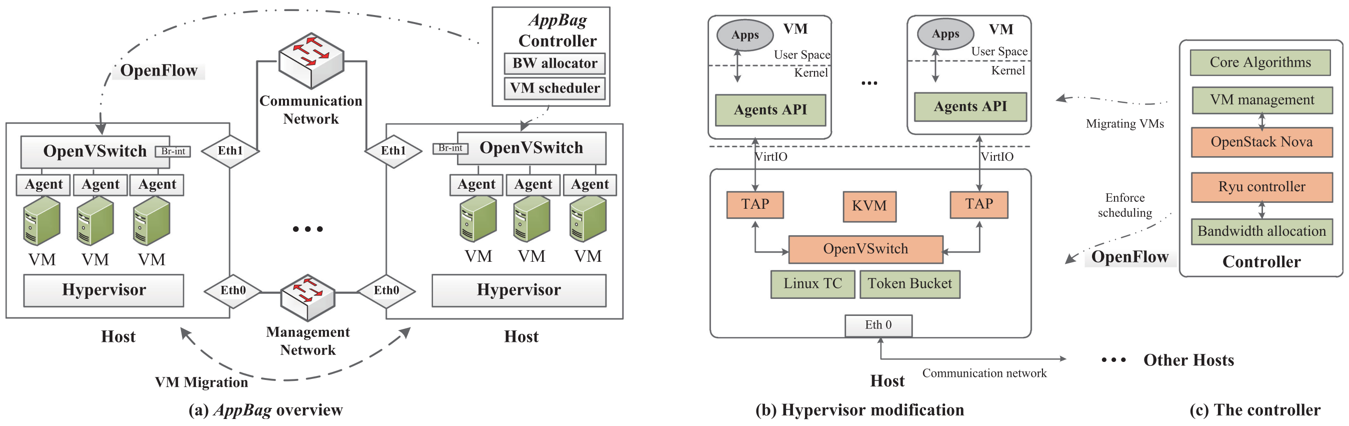


Fig. 3. The architecture and implementation of *AppBag*.

available network resources. Note that there is a complicated model to compute the threshold dynamically, considering both the distribution of flows and the capability of the controller, yet how to choose this threshold is beyond the scope of this paper. From our best practice, we set this value as 10-20 Mbps in this paper. After determining the bandwidth allocation, the controller communicates with hypervisors through the standard OpenFlow 1.3 protocol. The fields we use include *Matching fields*, *Instructions*, *Meters*, *Rate*, etc. VM live migrations are conducted through independent management network, without interfering the data communication of applications. Live migrations also have very little impact on the execution of applications, with the VM downtime as low as 30 ms [30].

Hypervisor Modification. In the *AppBag* architecture, each hypervisor is responsible for enforcing the bandwidth allocation for its hosting VMs. OpenVSwitch (OVS) [31] is installed on each hypervisor to integrate the VMs. The modification is implemented as an extension to OVS. It penetrates all VMs' outgoing packets through the Tunnel header to read the source and destination IP address. Querying the bandwidth allocated to each VM-to-VM pipe from the flow table, the packets are then delivered to Traffic Control (TC) for rate limiting. We leverage Token Bucket (TB) associated with the qdisc module to rate limit each pipe accordingly. TC modules require no kernel modifications and can be inserted and removed at run-time, making them flexible and easy to deploy. By design, *AppBag* uses fewer rate limiters than other implementations, making this solution scalable in the data center.

In Fig. 3, we depict a network where the entire data center fabric is abstracted out as one non-blocking switch interconnecting all the hypervisors. Because *AppBag* implements the rate limiters in the hypervisor (edge) layer, it is not sensitive to the data center topologies. In practise, *AppBag* would also perform well under various network topologies (e.g., the spine-leak topology).

Usage Scenario. In a public cloud environment, those tenants who need the guaranteed network performance choose *AppBag*-enabled images when launching VMs. The cloud provider encapsulates popular big data frameworks (e.g., Hadoop [20], Spark [32]) integrated with *AppBag* APIs into these VM images. The tenants automatically have a big data processing platform with predictable performance in the cloud. Alternatively, tenants are able to call the documented APIs to register their network demands at run-time in their specific programming framework. For the aspect of business model, the cloud providers charge the cost proportionally to the actual usage of network bandwidth. This incentivizes tenants to report their accurate network demands in the non-cooperative environment. For the tenants, they'd like to have predictable application performance and lower cost. For the providers, they can benefit from the improved utilization of infrastructure. The usage of *AppBag* is in consonance with current cloud computing business model and leads to a win-win situation.

4 Application-Aware Bandwidth Allocation

4.1 Problem Formulation

In this section, we first formulate the whole picture of application-aware bandwidth allocation through a rigorous mathematical model.

In the virtualization environment, the bandwidth allocation is to reserve the bandwidth to VMs and map them to the physical hosts with enough physical resources. Let h_x and H be the x^{th} physical host and the set of hosts within the data center, respectively. V and v_i denote the set of all VMs and the i^{th} virtual machine, respectively. With the reserved bandwidth, VMs are placed onto hosts with enough resources. The placement is captured by an $H \times V$ matrix denoted by P, where $p_{ix} \in P$ denotes whether v_i is hosted by h_x. Specifically, $p_{ix} = 1$ if v_i is hosted by h_x, otherwise $p_{ix} = 0$. For notational simplicity, we use one slot to represent one static resource unit (CPU/memory/disk), the size of v_i can be several slots, denoted as $s(v_i)$, while the size of h_x is $s(h_x)$. This simplification is also used by many previous works such as [33].

The VM's time-varying bandwidth requirements are specified using a set of traffic rules of the format [Timestamp, SrcVM, DstVM, traffic]. The rules are generated by the specific APIs invoked by the application. To capture the temporal variation, we first re-consolidate all VMs' traffic rules to one common time horizon. Define that the time stamp series for v_i is $0 \leq TS_i(1) \leq TS_i(2) \leq \ldots \leq TS_i(N_i)$, where N_i is the number of time stamps of v_i. If a VM varies its bandwidth requirement at time t, then we say that an event occurs at time t and t is called an event point. Therefore, from $t = 0$ to T, there are at most $\sum_{i=1}^{|V|} N_i$ events in total. As different VMs may alter the bandwidth

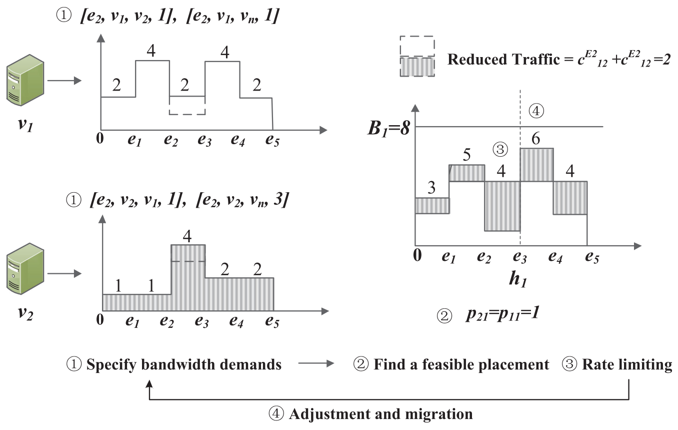

① [e2, v1, v2, 1], [e2, v1, vm, 1]
v1
① [e2, v2, v1, 1], [e2, v2, vm, 3]
v2
Reduced Traffic = c^E2_12 + c^E2_12 = 2
B1=8
h1
② p21=p11=1
① Specify bandwidth demands
② Find a feasible placement
③ Rate limiting
④ Adjustment and migration

Fig. 4. An example of bandwidth allocation and VM placement.

requirements at the same event point, the number m of distinct event points may be smaller, $m \leq \sum_{i=1}^{|V|} N_i$. We define that the m distinct event points $e_q, 1 \leq q \leq m$, are sorted in the order they occur, $0 \leq e_1 < e_2 < \ldots < e_m$. The time interval between two adjacent event points is called an epoch $E_q = [e_q, e_{q+1}), 1 \leq q \leq m-1$. The VM placement in epoch E_q is P^q.

We use the VM-to-VM pipe model to capture the spatial variation of VMs' communication traffic. The traffic rate between v_i and v_j in epoch E_q is denoted by $c_{ij}^{E_q}$, derived from the registered traffic rules. The traffic rates are further encoded in a $V \times V$ traffic matrix. Note that as the small-rate flows are ignored, the traffic matrix is actually very sparse. If two VMs are placed on the same host, the traffic goes through the local loopback of the host instead of traversing through the data center network. Therefore, the allocated bandwidth between v_i and v_j in E_q are defined as

$$b_{ij}^q = \begin{cases} 0 & \exists x, p_{ix}^q = p_{jx}^q = 1 \\ c_{ij}^{E_q} & otherwise \end{cases}. \tag{1}$$

Example. In Fig. 4, we illustrate an example of allocating bandwidth to two of VMs v_1 and v_2. By definition, there are 5 distinct event points e_1 to e_5 and 5 corresponding epoches E_0 to E_4. The aggregate bandwidth requirements in each epoch for v_1 and v_2 are $[2,4,2,4,2]$ and $[1,1,4,2,2]$, respectively. The unit of bandwidth is *Mbps*. In E_2, v_1 and v_2 has mutual communication of 1 unit, that is $c_{12}^{E_2} = c_{21}^{E_2} = 1$. At e_2, the applications first specify the bandwidth demands via the traffic rules, which are $[e_2, v_1, v_2, 1], [e_2, v_1, v_n, 1]$ and $[e_2, v_2, v_1, 1], [e_2, v_2, v_n, 3]$. Second, the system tries to find a feasible placement for v_1 and v_2, and determines to place them on physical host h_1, that is, $p_{11} = p_{21} = 1$. As v_1 and v_2 are co-resident, the traffic traversing the network is reduced by $c_{12}^{E_2} + c_{21}^{E_2} = 2$. The aggregate bandwidth allocated for v_1 and v_2 in E_2 is reduced to 4. Third, the system conducts rate limiting on v_1 and v_2 to enforce the bandwidth allocation. Fourth, when the network demands vary in the next epoch E_3, the system iteratively adjusts the placement for better allocations.

With the above definitions, the application-aware bandwidth allocation problem is formally stated as: given a number of VMs with bandwidth requests, in any epoch, trying to allocate the requested bandwidth and find a feasible placement P^* for these VMs onto physical hosts, so as to minimize the *cost* in the data center. Several constraints have to be satisfied: the aggregate bandwidth allocated to the VMs on each physical host should not exceed its physical bandwidth, the size of VMs should not exceed the physical host's capacity. This problem is formally defined as:

$$\begin{aligned} minimize & \sum_{q=0}^{m} g(P^q) + \beta \sum_{q=0}^{m} Impl(P^{q-1}, P^q) \\ s.t. & \sum_{i}^{|V|} \sum_{j}^{|V|} b_{ij}^q \cdot p_{ix} \leq B_x, \forall q, \forall x \\ & \sum_{i}^{|V|} s(v_i) \cdot p_{ix}^q \leq s(h_x), \forall q, \forall x. \end{aligned} \tag{2}$$

To apply this optimization, recall the major costs of VM bandwidth allocation: 1) the communication traffic among VMs in the data center network under a particular bandwidth allocation and placement; 2) the virtualization overhead incurred by migrating a VM from one host to another (or globally, switching from an old placement to a new one). We refer to the first part as "*communication cost*" and the second part as "*switching cost*" and consider a discrete-time optimization problem.

The *communication cost* in each epoch is modeled by $g(\cdot)$, which presents the global traffic (or the total bandwidth allocated) in the data center under a specific allocation. Authors in [33] have discussed the communication cost models of some specific applications. In this paper, we target at providing a more general model and define $g(\cdot)$ as:

$$g(P^q) = \sum_{i=1}^{|V|} \sum_{j=1}^{|V|} \sum_{x=1}^{|H|} \sum_{y=1}^{|H|} b_{ij}^q p_{ix} p_{jy}. \tag{3}$$

The *switching cost* from P^{q-1} to P^q is captured by $\beta \cdot Impl(\cdot)$, where β account for the cost of migrating a VM of 1 slot, and $Impl(\cdot)$ computes the aggregate size of migrated VMs.

$$Impl(P^{q-1}, P^q) = \sum_{i=1}^{|V|} \sum_{x=1}^{|H|} (p_{ix}^{q-1} \wedge p_{ix}^q) \cdot s(v_i). \tag{4}$$

In the above formulations, the value of β handles implicitly the cost of migrations and is usually difficult to decide in practice, we hereby resort to a detailed analysis of VM migration and provide a reference model to decide β.

Analysis of VM Migration. VM migration is a complicated process [30], [34], [35]. The VM migration cost differs with different underlying virtualization configuration, e.g., with/without shared storage. Without shared storage, it needs to transfer tens or hundreds GB of data stored in the local storage, and the cost is usually unacceptable. Instead, we use the shared storage configuration, where only the memory data need to be transferred in the migration. During the migration process, the hypervisor first copies all the memory pages from source to destination while the VM is still running. If some memory pages change (become "dirty") during this process, the hypervisor iteratively re-transfers the dirty pages. If the application has a high dirty

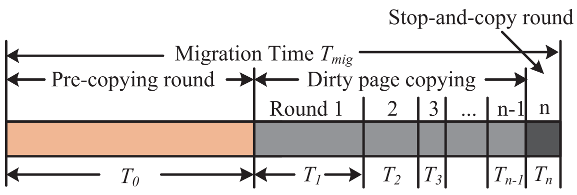


Fig. 5. VM migration in detail.

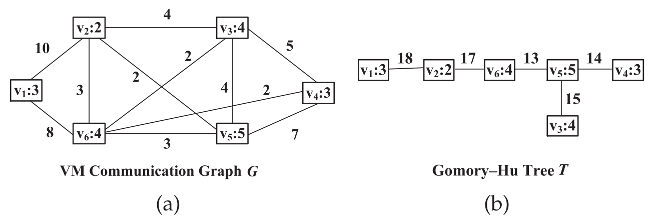


Fig. 6. One example of G and its associated Gomory-Hu tree.

rate, such "copy and re-transfer" iterations will take several rounds to complete. The migration procedure is shown in Fig. 5. For each migration, the migration time T_{mig} is related to the memory size M of 1 slot, transfer rate L, dirty rate D. Define $\lambda = D/L$, the migration time of each iteration T_n, T_{mig} is calculated by

$$T_{mig} = \sum_{0}^{n} T_i = \frac{M}{L} \cdot \frac{1-\lambda^{n+1}}{1-\lambda}. \tag{5}$$

In the realistic settings, T_{mig} usually has a timeout between 30s to 180s, and T_n is usually set around 30 ms, D can be measured experimentally. Substituting the values in Equation (5), we can derive the transfer rate L reserved to migrate 1 slot of VM. As the unit of communication cost is Mbps, we can set $\beta = L$. This choice of β unifies the communication cost and switching cost to the same dimension.

Given the rigorous mathematical formation of the application-aware bandwidth allocation problem, the goal is to minimize the cost. To solve this optimization with only one-step-ahead information, we first try to optimize the placement in each epoch to minimize the communication cost, and then extend the result to the whole time horizon with minimized switching cost.

4.2 Traffic-Aware VM Placement Problem in an Epoch

In this section, we first try to optimize the communication cost in each epoch. As there is little communication between applications, we find out that the cost optimization in any E_q is only affected by the applications which trigger the event point e_p. The global traffic matrix is then decomposed into individual ones for each application, our method follows an application by application manner to generate the allocation.

In each epoch E_q, given the traffic matrix of the applications which trigger the event point e_q, our algorithm aims to find an optimal solution which minimizes the communication traffic between hosts, satisfying the physical resource constraints. The problem can be modeled as a multi-constraints k-cut problem. In order to facilitate the analysis, we introduce some notations. Given an undirected graph $G = (V, E)$, the vertices in G are the VM set V of related applications. The communication between VMs are the edges E in G. The communication traffic c_{ij} is the weight of corresponding edge $w(e)$. Let C be a minimum k-cut in $G = (V, E)$ separating V into k components $V_1, V_2, ..., V_k$ and C_y be the section of cut separating V_y from $V - V_y$, we denote $w(V_y)$ as the weight of vertices, $w(C_y)$ as the weight of the cut. Given $r(B_x)$ as the residual bandwidth of h_x, the first constraint of Equation (2) is equal to

$$w(C_y) \le r(B_x), \forall x. \tag{6}$$

Let $r(h_x)$ be the residual slots in h_x, the second constraint of equation (2) is mathematically equal to

$$w(V_y) \le r(h_x), \forall x. \tag{7}$$

The minimum k-cut problem has been proved to be NP-hard and dynamic programming based algorithms are usually time-consuming. In order to solve it in a timely manner, we introduce the Gomory-Hu tree [36] representation of minimum cuts. Let T be a tree on vertex set V. Tree T is a Gomory-Hu tree for G if 1) for each pair of vertices $u, v \in V$, the weight of a minimum $u - v$ cut in G is the same as that in T; and 2) for each edge $e' \in T$, $w(e')$ is the weight of the cut associated with e' in G. A Gomory-Hu tree encodes, in a succinct manner, a minimum $u - v$ cut in G, and can be constructed using $n - 1$ maxflow computations. Fig. 6 shows a weighted graph and its associated Gomory-Hu tree.

With the property of Gomory-Hu tree, to generate n cuts in G, we need to remove $n - 1$ edges in T. As the weight of cut is equal to the bandwidth allocated, the heuristic is designed to greedily remove the edge with smallest weight, then place the generated two components (VM sets) into hosts by a first-fit manner. If the VM set can not fit into any host, the algorithm continues to cut the component until no further cut is needed. The algorithm VMgraphcut following a greedy and recursive manner is depicted in Algorithm 1.

Algorithm 1. VMgraphcut

```
1:  procedure VMgraphcut
2:      input: the Gomory-Hu tree $T$ of $G$, the connection weight
        with other parts $connect$
3:      output: a set of cuts $C$
4:      if $\exists x, w(T)$ fit $r(h_x)$ then
5:          return first-fit($w(T), r(h_x)$)
6:      end if
7:      find the smallest $e$ and cut $V$ into $V_1$ and $V_2$;
8:      if $w(e) + connect > \forall r(B_x)$ then
9:          return failure
10:     else
11:         VMgraphcut($V_1, w(e)$)
12:         VMgraphcut($V_2, w(e)$)
13:     end if
14: end procedure
```

The same as most other graph-cut algorithms, Algorithm 1 achieves an approximation factor of 2, which provides the upper bound for optimizing the communication cost in an epoch. For complexity, the most computation-intensive component of VMgraphcut is the generation of Gomory-Hu tree, which is of the polynomial complexity $O(|V|^3)$. In our evaluation, generating the Gomory-Hu tree for a $100 * 100$ traffic matrix takes about 150 microseconds, and for a emulated

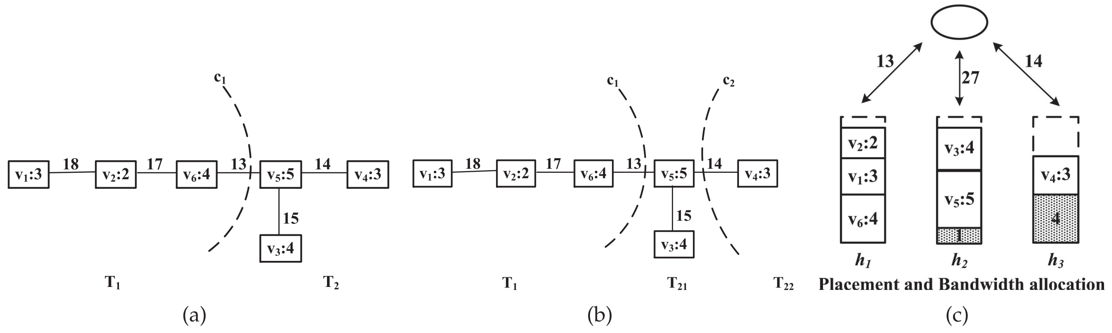


Fig. 7. One example of bandwidth allocation and VM placement for one application. The application is associated with its communication graph (Gomory-Hu tree representation). Each host has 30 units of bandwidth and 10 slots in total. The residual slots are of 10, 9, 6 for h_1, h_2, h_3, respectively.

$5000 * 5000$ traffic matrix, which is much more massive than that of general applications, takes less than 2.7 second. Fig. 7 gives an example of how to allocate the bandwidth and place the VMs.

4.3 VM Migration between Epoches

In this section, we dynamically adjust the VM placement to extend the optimization in one epoch to the future ones. Informally, the one-step-head bandwidth allocation problem is solved by, at the beginning of epoch e_q, optimizing the cost over the time horizon, using only the placement information P^{q-1} and P^q.

To transit from P^{q-1} to P^q, the system needs to conduct a set of migrations M, $|M| < |V|$, where migration $m_i \in M$ is VM v_i changing its physical host in P^{q-1} to the one in P^q. As m_i incurs the migration cost corresponding to β, their is an optimal solution $\mathbb{X}^*$ on M to equation (2) whether to conduct m_i. The goal of one-step-head bandwidth allocation is to find an online algorithms for this optimization.

Motivated by this goal, we present the analysis and implementation of a novel online algorithm, Lazy Migration. LM works by "lazily" staying within P^{q-1} and conducting only the most beneficial migrations to P^q. the migration decision $\mathbb{X}^{LM}$ generates the placement P^{LM} for the next epoch.

We first analyze the benefit of each migration m by modeling the cost savings possible from m. As P^q has a lower communication cost in E_q than P^{q-1}, every $m_i \in M$ is one step approaching the targeting low cost placement. Intuitively, the benefit of m_i can be interpreted as the value, in terms of communication cost, added to $g(P^q)$ if m_i is not conducted. Then we have:

Definition 1 Migration benefit. *The migration benefit b_{m_i} of migration m_i is defined as $b_{m_i} = g(P \ominus m_i) - g(P)$, where $\ominus$ defines the operation to revoke m.*

Analysis of Dependency. However, it turns out that to simply compute b_{m_i} for each m_i in this way is problematic. Let us consider the case in Fig. 8. If using the above definition, the benefit of individual migration b_{m_2} and b_{m_2} are 17 and 18 respectively, whereas the benefit of migrating v_2 and v_3 together b_{m_{2},m_3} is only 15. $b_{m_2} + b_{m_3} << b_{m_2,m_3}$, that is to say, b_{m_2} and b_{m_2} are highly exaggerated. Such exaggeration leads to a situation that multiple unnecessary migrations with wrongly high benefits are conducted, accounting for the high switching cost. Even worse, the system can be degraded into an oscillated state caused by these frequent and continuous migrations.

The root cause for this issue is the traffic dependency between v_2 and v_3. In the case illustrated in Fig. 8, v_2 and v_3 have the communication traffic of 10 units. By definition, the benefit gained by m_2 is computed by $g(P^q) - g(P^q \ominus m_2)$, which includes the 10 units of traffic between v_2 and v_3, while the same proportion of traffic savings is duplicately included by b_{m_3}. Therefore, the traffic dependency between v_2 and v_3 accounts for the exaggeration of individual b_{m_2} and b_{m_3}. Without loss of generality, we conclude that, transiting from P^{q-1} to P^q, m_i and m_j are mutually independent when and only when v_i and v_j have no communication traffic in e_q.

Motivated by this issue, we resort to grouping the migrations by dependency. By scanning the traffic among the corresponding v in M, independent migration groups $[m_i, m_j, ...]$, $[m_k, m_l, ...]$, ... are generated so that there is no communication traffic among these groups. For these migration groups, we first calculate their aggregate benefit, and contribute the benefit to individual migration proportionally to the weights $s(v)$.

Definition 2 Dependent migration benefit. *For a migration group $[m_i, m_j, ...]$, the migration benefit $b_{[m_i,m_j,...]}$ of*

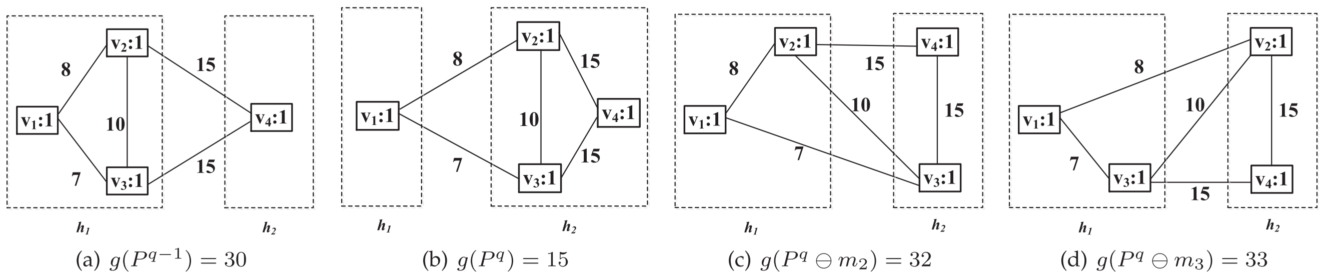


Fig. 8. One example of migration with dependency. By definition, the benefit of individual migration b_{m_2} and b_{m_2} are 17 and 18 respectively, whereas the benefit of conducting m_2 and m_3 together b_{m_2,m_3} is consequently 15. $b_{m_2} + b_{m_3} << b_{m_2,m_3}$.

dependent migrations $m_i, m_j, \ldots$ is calculated by $b_{[m_i,m_j,\ldots]} = g(P \ominus (m_i, m_j, \ldots)) - g(P)$, and b_{m_i} in the group is $b_{[m_i,m_j,\ldots]}/s(v_i)$.

Given the carefully defined benefit b_{m_i} of each m_i and its associated cost $\beta \cdot s(v_i)$, we are ready to design the LM algorithm, which solve the bandwidth allocation optimization with one-step-ahead information. In practice, LM works in an online manner to determine $x_i \in \mathbb{X}^{LM}$, whether to conduct m_i. The detail of LM is illustrated in Algorithm 2.

Algorithm 2. Lazy Migration

```
1:  procedure Lazy Migration
2:    input: the old placement $P^{q-1}$, the new placement $P^q$
3:    output: the optimal placement $P^{LM}$
4:    generate the independent migration groups of $M$
5:    for each $m$ in $M$ do
6:      maintains the cooresponding $x$ and $z_q$
7:      if $x < 1$ then
8:        $z_q = 1 - x$
9:        $x = x(1 + \frac{b_m}{\beta \cdot s(v_i)}) + \frac{1}{c \cdot \beta \cdot s(v_i)}$
10:       $y_q = 1$
11:     end if
12:     if $x \geq 1$ then
13:       conduct $m$ and generate $P^{LM}$
14:     end if
15:   end for
16: end procedure
```

We then try to analyze the performance of LM using the standard notion of competitive ratio. The competitive ratio of LM is defined as the supremum, taken over all possible inputs, of $cost(LM)/cost(OPT)$, where $cost(LM)$ is the objective function of optimization (2) under LM and $cost(OPT)$ is of the optimal algorithm.

Theorem 1. *For each m, LM produces a solution for optimization (2) of $\mathcal{O}(\frac{1}{c} + d)$-competitive, where $c = (1 + \frac{b_m^L}{\beta \cdot s(v)})^{\beta \cdot s(v)} - 1$, and $d = b_m^U$.*

Proof of Theorem 1 is in Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSC.2019.2922176. The analytic result provides the worst-case guarantee, where c and d are related with b_m^L and b_m^U. Note that in *AppBag*' realistic setting, both b_m^L and b_m^U are bounded. b_m^L is bounded by the minimum flow rate, as described in Section 3. Meanwhile, b_m^U is bounded by the physical bandwidth of PM, as it is the maximum bandwidth one VM can use. Although the competitive ratio is theoretically loose, but in practise this algorithm works well in improving the utilization of network infrastructure, and we experimentally evaluate its performance and the impact of b_m^L and b_m^U in Section 5.

LM is computationally inexpensive. For our 18-machine cluster, calculating the migration decision takes about 50 microseconds. Distributing the decision to all hypervisors takes about 4-5 milliseconds, and to 10000 (emulated) hypervisors takes less than 0.7 second. The run-time complexity to generate migration group is $O(|M|)$, as it needs to scan all migrations to check the dependency. Step (5) to (15) are one iteration of all migrations, it is of complexity $O(|M|)$. Therefore, the algorithm can be executed within $O(|M|)$. In practice, $|M|$ is bounded by the scale of the applications, and the algorithm can run very efficiently. Moreover, the result of Algorithm 2 is encouraging that it discards most of the unnecessary migrations, avoiding too much overhead and the system oscillation caused by frequent migrations.

5 Evaluation

In this section, we present an extensive testbed based evaluation of *AppBag*. The experiments are conducted in order to better understand:

1) how effective is *AppBag* in application-aware bandwidth allocation?
2) when is *AppBag* most beneficial in improving the performance of applications?

5.1 Experiments Setup

Testbed. The testbed is built using a simple leaf-spine architecture. There are a total of 18 servers connected to 2 leaf Top-of-Rack (ToR) switches with 1 Gbps links. Each leaf switch is connected to a spine switch using 10 Gbps links, ensuring full bisection bandwidth. The switches are configurated as standard output-queued switches. The server is equipped with two 6-core Xeon 2.66 GHz processors, 24 GB memory and a dedicated hard drive (a maximum of 12 VM slots). Each VM is configured with 2 GB-4 GB memory, 100 GB-200 GB hard disk and 1-2 dedicated CPU cores (1-2 VM slots). The server and network oversubscription ratio is set to be 1.

Workloads. The real-world workloads of applications we deployed are as follows:

RECOMM, a MapReduce application which has been studied in Section 2.1, generating the personal recommendations through data mining.

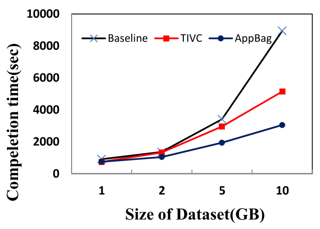
PAGER, as studied in Section 2.1, is a pregel-like graph processing application which computes the page-rank values for the input graph.

WEB, a typical Web service application for content querying, downloading, etc. WEB contains a pool of web-containers and a cluster of databases. A request is forwarded from the container to the database and then back to the client. Unlike RECOMM and PAGER, WEB represents another kind of popular applications existing in the cloud data center. Such kind of applications is classified as layered applications in [11]. The arrival of requests follow a Poisson process within $10 < \lambda_{web} < 100$.

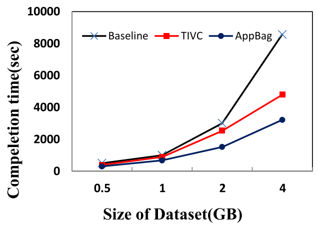
Background traffic, which is generated by VMs communicating with a varying number of random peers, in a uniform distribution between 5 and 20. The data exchanged between each pair of VMs is uniformly distributed between 1 and 100 Mbps.

Schemes Compared. We have implemented the following:

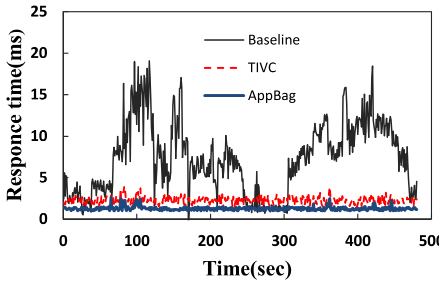
TIVC, the major comparable algorithm of *AppBag* is TIVC [10] with hose model (referred to as TIVC), because many existing methods are based on or variants of this typical scheme. In the hose model, N virtual machines are connected via a virtual switch with B-rate links. It means that a maximum rate of B is what one virtual machine can achieve. In our implementation of TIVC, when the



(a) Completion time of RECOMM.



(b) Completion time of PAGER.



(c) Response time of WEB.

Fig. 9. Comparison of applications' completion time.

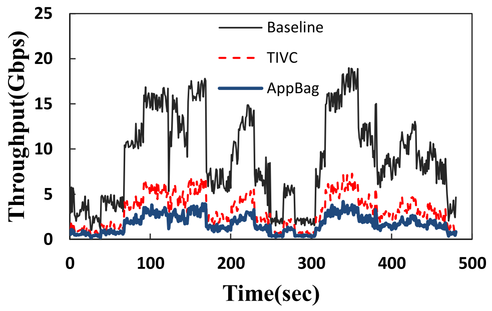


Fig. 10. Overall network throughput.

bandwidth requirements vary, it follows a first-fit manner to reallocate the VMs.

Baseline, which allocates the bandwidth for each VM statically according to their average network usage. We include the experiment results of Baseline, as a reference, to further explore where the benefits of our application-aware bandwidth allocation stem from.

AppBag, our design as described in Section 3.

5.2 Experiments Results

We compare *AppBag* with other schemes under two scenarios: (1) A number of applications have been scheduled to run in the data center, the operator tries to optimize the overall performance of these jobs (test case 1-5); (2) Application requests arrive dynamically and are accepted only when there are enough slots and residual bandwidth (test case 6-8). Some highlighted experiment results include:

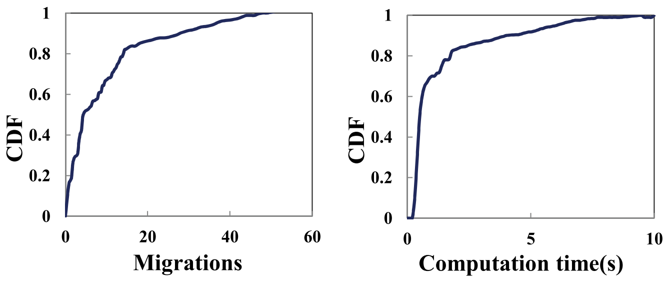
i) *AppBag* can reduce the execution time of applications by up to 47.3 percent and the global traffic by up to 36.7 percent.
ii) *AppBag* is capable of handling high volume of dynamically arriving applications. When $\lambda = 25$, the rejection rate is as low as 11.2 percent.
iii) The measured system occupation and network occupation are 24 and 17.5 percent lower than the state-of-the-art mechanisms, respectively.

5.2.1 Determinate Applications

In this scenario, we deploy RECOMM on 40 VMs and generate the datasets with the size of 1 GB, 2 GB, 5 GB and 10 GB, respectively. PAGER is deployed on 50 VMs and prepare the datasets with the size of 0.5 GB, 1 GB, 2 GB and 4 GB, respectively. To simulate a data center with diverse applications, we deploy WEB on 40 VMs and create 60 VMs which randomly generate background traffic. In the following test cases, we use the relative speed-up to represent the effectiveness of *AppBag*, which is defined as the amount of time that *AppBag* saved (or added) to the completion time of an application. For instance, if an application runs 10 minutes with TIVC, and 8 minutes using *AppBag*, the relative speed-up would be $(10 - 8)/10 = 20\%$.

Test Case 1 (*Execution Time*). One primary goal of *AppBag* is to provide the predictable performance for applications running in VMs. We compare the completion time of RECOMM and PAGER with different sizes of datasets under Baseline, TIVC and *AppBag*. For the WEB application, we continually test the response time in 500 seconds. The results are shown in Fig. 9. As expected, we observe that *AppBag* performs the best among other benchmarks. The bandwidth competition under Baseline and TIVC leads to poor and unpredictable performance of applications. On average, *AppBag* achieves 42.3 percent performance improvement of RECOMM than TIVC and 47.3 percent that of PAGER. The results on WEB also explicitly indicate the effectiveness of *AppBag* that when the traffic in the data center is high, the response time remains short and stable. The overall response time under *AppBag* is about 38.2 percent shorter than TIVC. Results in this test case demonstrate *AppBag*'s effectiveness in providing high and predictable performance with both the communication-intensive applications and traditional ones.

Test Case 2 (*Aggregate Traffic in the Data Center*). A collateral benefit of *AppBag* is that the communication traffic in the data center fabric is reduced by a large amount. The traffic among co-resident VMs should go through the local loopback of the host, saving the inter-host bandwidth. In this test case, we measure the egres/ingress traffic of each physical host and aggregate them as the total traffic. The results are shown in Fig. 10. Baseline, without dynamically adjusting the resource allocation, demonstrates the original traffic pattern in the data center, which presents a significant fluctuation. TIVC, adaptive to the fluctuation, reduces the total traffic to some extent. *AppBag*, with the accurate traffic information among VMs, achieves an average 36.7 percent traffic saving than TIVC. Especially under heavy traffic, *AppBag* migrates the VMs with the most intensive communication to the same physical host. For example, between 72 sec to 119 sec of the evaluation, *AppBag* reduce the aggregate traffic by 49.3 and 83.6 percent when compared with TIVC and Baseline. Between 313 sec to 3180 sec, it reduces the traffic by 42.2 and 85.9 percent. It is obvious that *AppBag* is able to utilize the network more efficiently and enables data centers to accept more applications in potential, and we evaluate this capability in detail in Section 5.2.2.

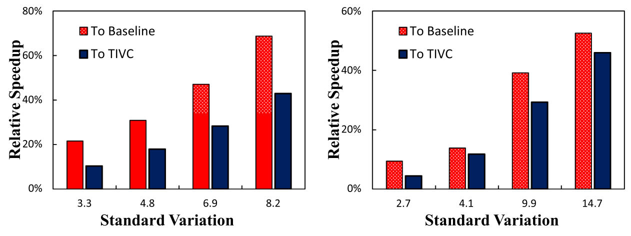


(a) Migration performed between the epoches. (b) Computation time of one decision.

Fig. 11. System overhead.

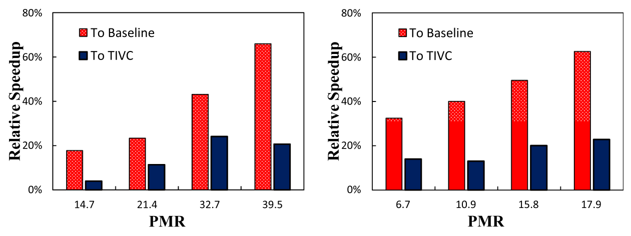
Test Case 3 (*Overhead*). The overhead incurred by computation and virtualization is the key to the practicality and scalability of *AppBag*. In previous test cases, we record the computation time to make allocation decisions and the number of migrations between epoches. The results are shown in the CDF (Cumulative Distribution Function) figures in Fig. 11. We observe from Fig. 11b that the computation overhead is acceptable that more than 80 percent of placement can be derived in 1.4 seconds. This is achieved by the multiple optimizations we made in the design and implementation of *AppBag*. In the aspect of VM migrations, 80 percent of the new placements are achieved through less than 14 migrations. The reason behind this is that *AppBag* exploits the lazy migration strategy by conducting only the most beneficial migrations, avoiding system oscillation. Furthermore, all the overhead of *AppBag* is subject to the VM scale of the individual application, instead of the total number of VMs in the data center. As the VM scale of individual application is substantially smaller and quite limited, *AppBag* is practical and scalable to deploy in production data centers.

Test Case 4 (*The Impact of Spatial Variation*). To quantify spatial variability, let F_{ij} be the fraction of the application's traffic sent from VM_i to VM_j. In the ideal case, every intra-application connection sends the same amount of data at a constant rate, the distribution of these F values has a standard deviation of zero. We adjust the size and data distribution of the datasets of RECOMM and PAGER to generate different F, compute the standard deviation of F values and then plot the relationship between spatial variation and relative speed-up. From Fig. 12, for both applications, *AppBag* achieves larger speed-up when spatial variation increases. This is also true when compared with TIVC model, where *AppBag* can achieve an average 32.5 percent speed-up and a maximum of 43.5 and 45.3 percent speed-up for RECOMM and PAGER, respectively. When comparing to Baseline,



(a) Speed-up of RECOMM. (b) Speed-up of PAGER.

Fig. 12. The impact of spatial variation.



(a) Speed-up of RECOMM. (b) Speed-up of PAGER.

Fig. 13. The impact of PMR.

AppBag is able to achieve as much as 66.9 and 57.1 percent acceleration for RECOMM and PAGER with high spatial variation. This improvement helps us to explore the promising optimization space of dynamic bandwidth allocation. As the spatial variation of traffic is a common characteristic of popular applications in data centers, *AppBag* is competitive in this context.

Test Case 5 (*The Impact of Peak-to-Mean Ratio*). We first introduce the peak-to-mean ratio (PMR) of the traffic patterns. *PMR=max(traffic)/mean(traffic)* measures the traffic variation of the application. Larger *PMR* indicates that the application experiences more dramatic traffic change. What is worth noticing is that, although *PMR* restricts the theoretical bound of the core algorithm, *AppBag* experimentally proves its effectiveness, especially when *PMR* is large. From Fig. 13, it is shown that *AppBag* outperforms other methods when *PMR* increasing. It is also obvious with a more than 60 percent improvement compared with Baseline for both applications. The time-varying methods of network allocation such as TIVC and *AppBag* are most beneficial when *PMR* is large. *AppBag* further outperforms TIVC for a large amount by avoiding frequent VM migrations.

5.2.2 Dynamic arriving applications

We now consider the scenario when application requests (VMs) arrive dynamically over time. There are 10 concurrent application requests and their arrival follows a Poisson distribution with λ per 30 mins. The size of applications is random between [1, N] slots, where N is chosen to be 48 here. If an application cannot be allocated enough slots upon arrival, it is then rejected, as is the case with most cloud providers' admission control (e.g., Amazon EC2). In this scenario, we mainly focus on evaluating *AppBag*'s impact on rejection rate, system utilization and network utilization.

Test Case 6 (*Rejection Rate*). Figs. 14a, 14b, 14c plot the rejection rate for the two application workloads and their mixed workload under different schemes. We observe that under low load, e.g., $\lambda = 1$ and $\lambda = 5$, the resource reservation under *AppBag*, TIVC and Baseline can be met and hence they all accept all requests. As λ increases, *AppBag* rejects far less requests than other schemes. For example, when $\lambda = 25$, on average 18.8, 11.2 and 16.5 percent of the application requests are rejected under *AppBag* compared to 22.1, 16.9, and 21.4 percent under TIVC, and 60.2, 51.6, and 65.5 percent under Baseline, for the three kinds of workloads, respectively. This indicates that, with accurate bandwidth allocation, *AppBag* is able to reduce the completion time of applications and enhance the utilization of infrastructure, thus allowing more requests to the system.

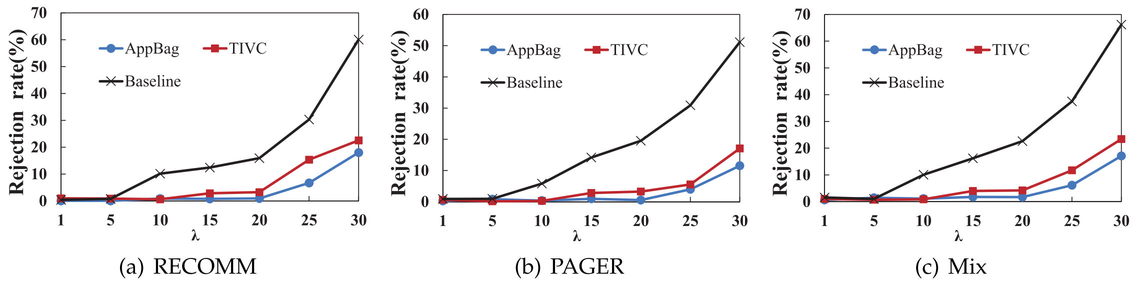


Fig. 14. Rejection rate of applications.

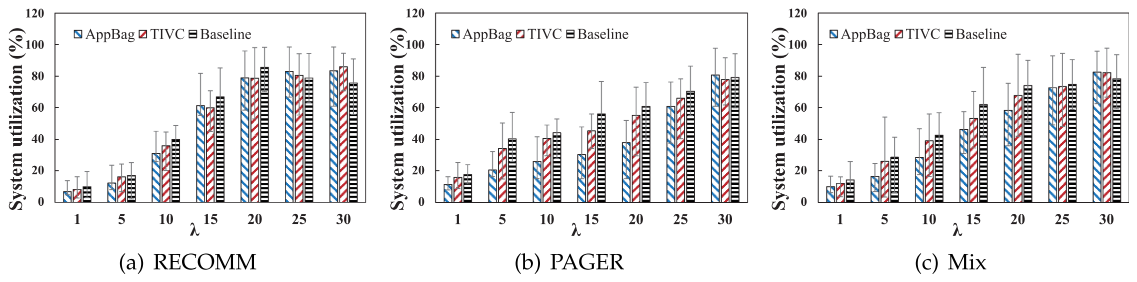


Fig. 15. System utilization of applications.

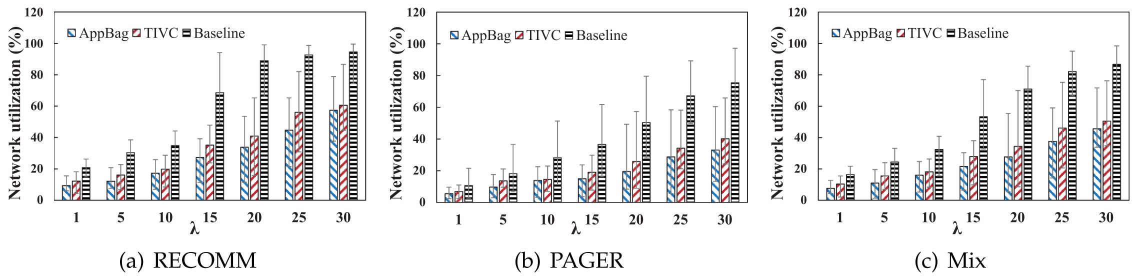


Fig. 16. Network utilization of applications.

Test case 7 (*System Utilization*). We then try to understand where the lower rejection rates of *AppBag* stem from. We first look at the system utilization. In each test case, we record the average utilization (bar height), associated with the 90th percentile utilization (upper bound of whiskers) and the 10th percentile utilization (lower bound of whiskers). As is shown in Fig. 15, when $\lambda \in [1, 15]$, the requests arrival ramps up and the system utilization increases steadily. In these cases, *AppBag* consistently achieves about 15 and 24 percent higher resource reservation. When $\lambda \geq 20$, the system utilization under *AppBag* and TIVC are approaching the bottleneck that the 90th percentile utilization of them are 99.3 and 94.3 percent. It means that there are few slots for VMs to be accepted. Whereas, the system utilization under Baseline fluctuates while the rejection rates keep increasing, indicating that although there are available slots, the system can not fulfill their network demands.

Test Case 8 (*Network Utilization*). It is obvious from Fig. 16a, 16b, 16c that, the three workloads encounter the network bottleneck under Baseline when $\lambda \geq 20$. On the contrary, under the time varying bandwidth schemes, network is no longer the bottleneck. Overall, *AppBag* outperforms TIVC by 14.7 percent in terms of the 90th percentile bandwidth reservation and by 17.5 percent in that of average reservation. Furthermore, when λ growing, there is a trend that the advantage of *AppBag* is more obvious. This confirms that by capturing the accurate traffic demand of applications, *AppBag* is able to achieve more efficient bandwidth reservation. The reserved bandwidth is useful for the diverse and bursty traffic of real world applications in data centers.

6 Related Work and Future Work

Data Center Bandwidth Allocation. Bandwidth allocation in the data centers has been a hot research topic, and there have been a flurry of works targeting at various objectives. Many aim at the flow completion time minimization [37], [38], fast convergence [29] or coflow scheduling [25], [39]. Nevertheless, the bandwidth allocation problem becomes more complicated with the virtualization technologies deployed in the modern data centers [1], [40], [41]. Existing designs have tried to tackle one or two points in the bandwidth allocation design space. Our previous paper [8] has demonstrated some promises in the direction of application-aware bandwidth allocation, while its capability to deal with more complicated cases are still not fully utilized.

Network Model. Oktopus supported the VOC model [12], which intended to match a communication pattern where clusters of VMs required high intra-cluster bandwidth and lower inter-cluster bandwidth. SecondNet [13] proposed virtual data center (VDC) as the unit of resource allocation for multiple tenants. The authors in [23] aimed at providing tenants with minimum bandwidth guarantees while bounding their maximum network impact. In [33], the authors analyzed the network cost functions under various communication models. CloudMirror [24] proposed TAG to capture the layered network model of applications.

Traffic Forecasting. Proteus [10] captured the time-varying bandwidth requirements of applications. Cicada [9] proposed a prediction based method to provide bandwidth guarantee to a tenant. ElasticSwitch [14] provided bandwidth guarantees for VMs and ensured work-conserving. In [17], the authors proposed application layer traffic forecasting through log files and meta-data, which can achieve nearly 100 percent forecasting accuracy.

VM Management. Traffic-ware VM management is close related to bandwidth allocation. This problem was first modeled by [42] that VMs with large mutual bandwidth usage are assigned to physical machines in close proximity. [33] further addressed the tradeoff between VM communication cost and physical resource utilization. [43] optimized jointly VM placement and routing to achieve higher scalability. [35] addressed the problem of network allocation for concurrent VM migrations.

To our best knowledge, *AppBag* is the first one to explore the bandwidth allocation problem by leveraging only one-step-ahead network demands of applications, and it provides a practical solution for current data centers. As for future work, we consider the following two:

Decentralized Framework. While the centralized design of *AppBag* can facilitate fast convergence to the global optimal allocation, the distributed framework has the advantages in scalability and reliability. It is still controversial to determine which is better. We believe that to design a decentralized version of *AppBag* is an important future work, which will be relevant for sharing wide-area networks in the context of geo-distributed data centers.

Coflow-Aware Bandwidth Allocation. In this paper, we have explored some preliminary properties of VMs' traffic dependency. Recent works have proposes the *coflow* [25], [39] semantics of such traffic dependency, where a collection of parallel flows needs to be transferred between groups of servers. We believe that there is still optimization space when taking consideration of coflows' properties. We leave exploring this area as a promising future work.

7 Conclusion

In conclusion, this paper has provided an in-depth analysis on application-aware VM bandwidth allocation problem with only one-step-ahead information. We explore the design space and propose *AppBag*, a practical framework which enables applications to specify their one-step-ahead traffic demands, and allocates the bandwidth to VMs at run-time. Key algorithms, namely VMGraphcut and Lazy Migration, are proposed and theoretically analyzed. The algorithms are easy to implement and they do not incur high computational overhead. Extensive evaluations are conducted, demonstrating that *AppBag* benefits both applications and operators with predictable performance and higher network utilization. A prototype system is implemented based on OpenStack, and it is practical and readily-deployable in real-world data centers.

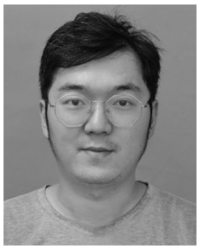
Acknowledgments

This work is supported by National Key R & D Program of China 2017YFB1003000, National Natural Science Foundation of China under Grants, No.61572129, No.61602112, No.61502097, No.61702096, No.61320106007, No. 61632008, International S & T Cooperation Program of China No. 2015DFA10490, Jiangsu Provincial Key Laboratory of Network and Information Security under Grants No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grants No.93K-9, University of Wollongong's UIC International Links Grant Scheme, and Global Challenge Program Travel fund, and partially supported by Collaborative Innovation Center of Novel Software Technology and Industrialization and Collaborative Innovation Center of Wireless Communications Technology. Fang Dong is the corresponding author of this paper.

References

[1] "Amazon Elastic Compute Cloud." [Online]. Available: http://aws.amazon.com/ec2/
[2] H. Luo, X. Liu, et al., "Where to fix temporal violations: A novel handling point selection strategy for business cloud workflows," in *Proc. IEEE Int. Conf. Serv. Comput.*, 2016, pp. 155–162.
[3] Y. Wang, Q. He, and Y. Yang, "Qos-aware service recommendation for multi-tenant saas on the cloud," in *Proc. IEEE Int. Conf. Serv. Comput.*, 2015, pp. 178–185.
[4] I. Petri, J. Diaz-Montes, et al., "Modelling and implementing social community clouds," *IEEE Trans. Serv. Comput.*, vol. 10, no. 3, pp. 410–422, May/Jun. 2017.
[5] X. Li, L. Zhang, et al., "A novel workflow-level data placement strategy for data-sharing scientific cloud workflows," *IEEE Trans. Serv. Comput.*, vol. 12, no. 3, pp. 370–383, May/Jun. 2019.
[6] L. Wang and J. Shen, "Multi-phase ant colony system for multi-party data-intensive service provision," *IEEE Trans. Serv. Comput.*, vol. 9, no. 2, pp. 264–276, Mar./Apr. 2016.
[7] J. C. Mogul and L. Popa, "What we talk about when we talk about cloud network performance," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, pp. 44–48, 2012.
[8] D. Shen, J. Luo, et al., "Appbag: Application-aware bandwidth allocation for virtual machines in cloud environment," in *Proc. 45th Int. Conf. Parallel Process.*, 2016, pp. 21–30.
[9] K. LaCurts, J. C. Mogul, et al., "Cicada: Introducing predictive guarantees for cloud networks," in *Proc. 6th USENIX Conf. Hot Top. Cloud Comput.*, 2014, p. 14.
[10] D. Xie, N. Ding, et al., "The only constant is change: Incorporating time-varying network reservations in data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, pp. 199–210, 2012.
[11] J. Lee, M. Lee, et al., "Cloudmirror: Application-aware bandwidth reservations in the cloud," in *Proc. HotCloud*, 2013.
[12] H. Ballani, P. Costa, et al., "Towards predictable datacenter networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 242–253, 2011.
[13] C. Guo, G. Lu, et al., "Secondnet: A data center network virtualization architecture with bandwidth guarantees," in *Proc. 6th Int. Conf.*, 2010, Art. no. 15.
[14] L. Popa, P. Yalagandula, et al., "Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 351–362, 2013.
[15] H. Lin, X. Qi, et al., "Workload-driven vm consolidation in cloud data centers," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 207–216.
[16] S. Zhang, Z. Qian, et al., "Burstiness-aware resource reservation for server consolidation in computing clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 4, pp. 964–977, Apr. 2016.
[17] Y. Peng, K. Chen, et al., "Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing," in *Proc. IEEE Conf. Comput. Commun.*, 2014, pp. 19–27.
[18] K. Nagaraj, D. Bharadia, et al., "Numfabric: Fast and flexible bandwidth allocation in datacenters," in *Proc. ACM SIGCOMM*, 2016, pp. 188–201.

[19] M. Chowdhury, Z. Liu, et al., "Hug: Multi-resource fairness for correlated and elastic demands," in *Proc. 13th Usenix Conf. Networked Syst. Des. Implementation*, 2016, pp. 407–424.
[20] "Hadoop Distributed File System." [Online]. Available: http://hadoop.apache.org/
[21] "MovieLens." [Online]. Available: https://grouplens.org/datasets/movielens/
[22] G. Malewicz, M. H. Austern, et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD*, 2010, pp. 135–146.
[23] H. Ballani, K. Jang, et al., "Chatty tenants and the cloud network sharing problem," in *Proc. 10th USENIX Conf. Networked Syst. Des. Implementation*, 2013, pp. 171–184.
[24] J. Lee, Y. Turner, et al., "Application-driven bandwidth guarantees in datacenters," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 467–478, 2014.
[25] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 443–454, 2014.
[26] H. Zhang, L. Chen, et al., "Coda: Toward automatically identifying and scheduling coflows in the dark," in *Proc. ACM SIGCOMM*, 2016, pp. 160–173.
[27] "OpenStack Open Source Cloud Computing Software." [Online]. Available: https://www.openstack.org/
[28] Y. Zhao, K. Chen, et al., "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 424–432.
[29] J. Perry, H. Balakrishnan, and D. Shah, "Flowtune: Flowlet control for datacenter networks," in *Proc. 14th USENIX Conf. Networked Syst. Des. Implementation*, 2017, pp. 421–435.
[30] C. Clark, K. Fraser, et al., "Live migration of virtual machines," in *Proc. 2nd Conf. Symp. Networked Syst. Des. Implementation* - Vol. 2, 2005, pp. 273–286.
[31] B. Pfaff, J. Pettit, et al., "The design and implementation of open vswitch," in *Proc. 12th USENIX Conf. Networked Syst. Des. Implementation*, 2015, pp. 117–130.
[32] "Apache Spark." [Online]. Available: http://spark.apache.org/
[33] X. Li, J. Wu, et al., "Let's stay together: Towards traffic aware virtual machine placement in data centers," in *Proc. IEEE Conf. Comput. Commun.*, 2014, pp. 1842–1850.
[34] T. Wood, P. J. Shenoy, et al., "Black-box and gray-box strategies for virtual machine migration," in *Proc. 4th USENIX Conf. Networked Syst. Des. Implementation*, 2007, p. 17.
[35] H. Wang, Y. Li, et al., "Virtual machine migration planning in software-defined networks," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 487–495.
[36] D. Panigrahi, "Gomory-hu trees," in *Encyclopedia of Algorithms*. New York, NY, USA: Springer, 2008, pp. 364–366.
[37] M. Alizadeh, S. Yang, et al., "pfabric: Minimal near-optimal datacenter transport," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 435–446, 2013.
[38] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 127–138, 2012.
[39] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 393–406, 2015.
[40] K. He, E. Rozner, et al., "Ac/dc tcp: Virtual congestion control enforcement for datacenter networks," in *Proc. ACM SIGCOMM*, 2016, pp. 244–257.
[41] B. Cronkite-Ratcliff, A. Bergman, et al., "Virtualized congestion control," in *Proc. ACM SIGCOMM*, 2016, pp. 230–243.
[42] 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.
[43] Y. Zhao, Y. Huang, et al., "Joint vm placement and topology optimization for traffic scalability in dynamic datacenter networks," *Comput. Netw.*, vol. 80, pp. 109–123, 2015.

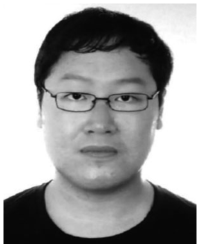


Dian Shen received the bachelor's, master's, and PhD degrees from Southeast University, China, in 2010, 2012, and 2018, respectively. He is an assistant professor with the School of Computer Science and Engineering, Southeast University, China. His research interests include cloud computing, virtualization and data center network.

Junzhou Luo received the BS degree in applied mathematics from Southeast University, in 1982, and the MS and PhD degrees in computer network both from Southeast University, in 1992 and 2000, respectively. He is a full professor with the School of Computer Science and Engineering, Southeast University, China. His research interests include network security and management, cloud computing, and wireless LAN. He is a member of the IEEE and ACM, and co-chair of IEEE SMC Technical Committee on Computer Supported Cooperative Work in Design.

Fang Dong received the BS and MS degrees in computer science from Nanjing University of Science and Technology, China, in 2004 and 2006, respectively, and the PhD degree in computer science from Southeast University, in 2011. He is currently an associate professor with the School of Computer Science and Engineering, Southeast University, China. His current research interests include cloud computing, task scheduling and big data processing.

Jiahui Jin received the PhD degree in computer science from Southeast University, in 2015. He had been a visiting PhD student at the University of Massachusetts, Amherst, during August 2012 to August 2014. He is an assistant professor with the School of Computer Science and Engineering, Southeast University, Nanjing, China. His current research consists of large-scale data processing, distributed systems, and parallel task scheduling.



Junxue Zhang received the bachelor's degree in software engineering and the master's degree in computer science and engineering from Southeast University, China, in 2013 and 2016, respectively. He is currently working toward the PhD degree in CSE, Hong Kong University of Science and Techonology. His current interest includes data center networking.

Jun Shen is an associate professor with the University of Wollongong, Wollongong, NSW, Australia. His expertise is on web services and Semantic Web. He has been an editor, PC chair, guest editor, and a PC member for numerous journals and conferences published by the IEEE, ACM, Elsevier, and Springer. From 2007, he was a chair of Education Chapter of IEEE NSW section. He is a senior member of the IEEE.