Application-Driven Bandwidth Guarantees in Datacenters

Jeongkeun Lee Yoshio Turner Myungjin Lee* Lucian Popa+ Sujata Banerjee
Joon-Myung Kang Puneet Sharma
HP Labs, *University of Edinburgh, +Databricks

ABSTRACT

Providing bandwidth guarantees to specific applications is becoming increasingly important as applications compete for shared cloud network resources. We present *CloudMirror*, a solution that provides bandwidth guarantees to cloud applications based on a new network abstraction and workload placement algorithm. An effective network abstraction would enable applications to easily and accurately specify their requirements, while simultaneously enabling the infrastructure to provision resources efficiently for deployed applications. Prior research has approached the bandwidth guarantee specification by using abstractions that resemble physical network topologies. We present a contrasting approach of deriving a network abstraction based on application communication structure, called *Tenant Application Graph or TAG*. CloudMirror also incorporates a new workload placement algorithm that efficiently meets bandwidth requirements specified by TAGs while factoring in high availability considerations. Extensive simulations using real application traces and datacenter topologies show that CloudMirror can handle 40% more bandwidth demand than the state of the art (e.g., the Oktopus system), while improving high availability from 20% to 70%.

Categories and Subject Descriptors: C.2.3 [Computer-Communication Networks]: Network Operations
Keywords: Datacenter; Bandwidth; Availability; Cloud; Virtual Network; Application

1. INTRODUCTION

A growing trend is the consolidation of computing infrastructure and applications into large datacenters, including virtualized cloud environments. Many of these emerging cloud applications are complex combinations of multiple services and require predictable performance, high availability, and high intra-datacenter bandwidth; e.g., Facebook "experiences 1000 times more traffic inside its data centers than it sends to and receives from outside users", and the internal traffic has increased much faster than Internet-facing bandwidth [1]. Meanwhile, many datacenter networks are oversubscribed, as high as 40:1 in some Facebook datacenters [2], causing the intra-datacenter traffic to contend for core bandwidth. Hence, *providing bandwidth guarantees* to specific applications is highly desirable, in order to preserve their response-time predictability when they compete for bandwidth with other applications.

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2619239.2626326.

Today, it is easy to share and virtualize compute and storage resources effectively. In contrast, implementing network virtualization with bandwidth guarantees on a shared network infrastructure is an inherently complex and challenging task [3–7, 18, 45], which requires three key technologies: 1) An easy-to-use *network abstraction model* for tenants to accurately express their bandwidth requirements; 2) A workload *placement algorithm* that efficiently allocates datacenter resources to meet the tenant requests, and 3) A scalable *runtime mechanism to enforce the bandwidth guarantees* and utilize unused bandwidth efficiently. In this paper, we propose *CloudMirror*, a solution that combines a new network abstraction with a new workload placement algorithm, while leveraging an existing mechanism [7] for enforcing guarantees.

An effective network abstraction model serves two purposes. One purpose is for tenants to specify their network requirements in a simple and intuitive yet accurate manner. The other purpose is to facilitate easy translation of these requirements to an efficient deployment on the low level infrastructure components. Most prior work, e.g., [4–9], has designed abstractions for specifying bandwidth guarantees that can be expressed as idealized physical network models, e.g., non-blocking switch (hose) [8] or two-level tree (hierarchical hose) [4, 6]. This is a natural approach since, for example, in cloud computing, tenants want to have the illusion of running their applications on dedicated hardware; with such a model, tenants would have the illusion of running their application over a dedicated physical network.

In contrast to these prior approaches, our approach is *to derive the network abstraction model based on application communication structure, and not a given underlying physical network topology.* We show that not only is such an abstraction easier to understand and reason about by tenants, but it can also be significantly more efficient in reserving bandwidth than the commonly proposed abstractions, such as the hose model. The intuition for its efficiency is that our abstraction accurately captures the bandwidth requirements of an application rather than imposing a pre-defined and perhaps a poor fit network abstraction (e.g., hose) for applications to map their requirements to.

To instantiate the bandwidth guarantees, the high-level abstraction must be mapped onto the low level physical topology via a job (VM) placement mechanism. We describe a new algorithm that exploits our refined network abstraction to more efficiently utilize datacenter resources. Efficient bandwidth utilization is often achieved by colocating application VMs in a single server or rack, which hurts availability; our placement algorithm provides high availability (HA) while efficiently guaranteeing bandwidth.

Finally, a runtime mechanism must enforce the virtual network abstraction for any traffic matrix in the datacenter. Our network abstraction can be easily supported through minor changes to existing frameworks for enforcing hose guarantees, such as those proposed in [4, 5, 7].

In this paper, we make the following contributions, building on our prior work [10].

1. A new tenant network abstraction model – *Tenant Application Graphs or TAGs* (§3). Deriving application structures from empirical datasets, we show that TAGs can be easier to use and more efficient in reserving bandwidth than the existing abstractions. We also develop a methodology to generate TAG models automatically from raw communication traces.
2. A new fast VM placement algorithm that resource-efficiently maps TAGs onto a tree-shaped physical network, satisfying the bandwidth requirements and also any specified high availability (HA) goals (§4). Bandwidth efficiency and HA can be conflicting goals [11]. We mathematically derive conditions that determine when colocating VMs would save bandwidth resource. When the conditions are violated, i.e., no bandwidth savings from colocation, our VM placement adopts anti-colocation to improve HA and to achieve balanced utilization of bandwidth with other types of resources.
3. Demonstration of the benefits of the TAG model and of our placement algorithm through extensive simulations using real application traces and datacenter topologies (§5). With a simple prototype, we show that the TAG model can be easily implemented on top of an existing mechanism [7] that enforces the hose model.

We next describe the current state of the art in providing bandwidth guarantees in cloud datacenter networks. Through examples of real applications, we show how existing network abstractions fall short and thus motivate the need for our new TAG model.

2. BACKGROUND AND RELATED WORK

Prior cloud networking research has primarily focused on supporting network requirements for Hadoop and Pregel like batch processing applications. Obviously, many applications are not similar in structure to Hadoop or Pregel exhibiting simple all-to-all communication patterns. In this paper we focus on other application types such as interactive applications (e.g., web and OLTP) hosted in today's cloud environments [12–14]. These applications have complex and tiered structures, and are not well represented using the prior models (§2.2). Moreover, unlike batch applications that can tolerate network bottlenecks, *interactive applications have very stringent performance requirements, and demand predictable throughput and tail latencies* [13]. Amazon has reported incurring e-commerce sales loss of 1% for every 100 msec increase in response latency [15]. Parley [16] demonstrated that adequate bandwidth is critical for applications to maintain low tail latencies. In our tests with Wikipedia benchmark [17], we also observed a sharp increase in web response time (from 250 to 900 msec) when the network bandwidth between the web and database VMs was throttled, only for 10 seconds, to 10% below bottleneck-free capacity.[1] This demonstrates the severe impact of insufficient bandwidth on web applications and a strong need for guaranteeing bandwidth for interactive applications.

Our private conversations with cloud users/operators as well as various benchmark reports confirm that *non-batch, interactive workloads often have similar or higher bandwidth requirements relative to more CPU-bound batch workloads*. Fig. 1(a) plots the ratio of aggregate application throughput (Mbps) to aggregate CPU consumption (GHz) for various cloud workloads.[2] From

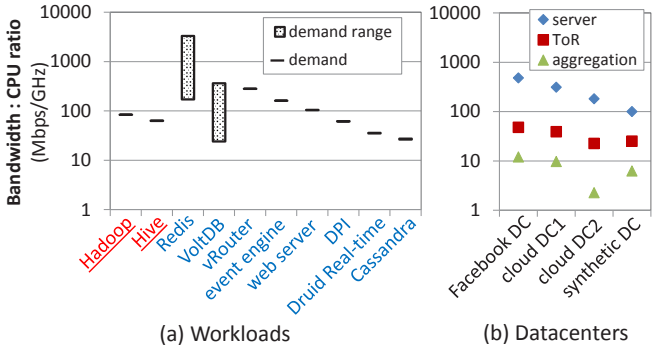


Figure 1: Bandwidth-to-CPU ratio for 10 workloads and 4 datacenters. Batch jobs in red; interactive applications in blue.

Fig. 1(a), we observe that the interactive workloads (Redis to Cassandra [19–24]) have similar or higher ratios of network-to-CPU compared to the batch jobs (Hadoop and Hive [18]).

Meanwhile, today's datacenters (DCs) are often oversubscribed and lack enough bandwidth to avoid contention between applications. Fig. 1(b) plots the provisioned ratio of bandwidth-to-CPU resources of four cloud datacenter environments at different tree topology levels.[3] We consider two production cloud DCs, Facebook DC [2,25] and the synthetic DC topology simulated in [4,18]. Comparing Figs. 1(a) and 1(b), we find that most datacenters are well provisioned to meet network demands of the workloads at the server level, but not at the ToR or aggregation level due to network oversubscription. Despite a recent trend toward less oversubscription, provisioning full-bisection bandwidth remains costly for large-scale datacenters. Bandwidth contention on oversubscribed core links is worsened by the need to spread VMs of an application across multiple servers and racks to increase robustness to single-point-of-failure. Even for non-oversubscribed networks, an efficient tenant network abstraction coupled with bandwidth guarantees benefits applications and operators, because extra bandwidth can facilitate lossless/fast network updates [26] and fault-resiliency [27].

While some cloud providers start to guarantee bandwidth [28, 29], they do so at specific fixed bandwidth-to-vCPU ratios, which limits flexibility in serving applications with diverse bandwidth-to-CPU demand ratios (Fig 1(a)). Their models, typically simplified from the hose model, also fail to capture bandwidth demands in an accurate or resource-efficient way, as we describe next.

2.1 Example Application Structure

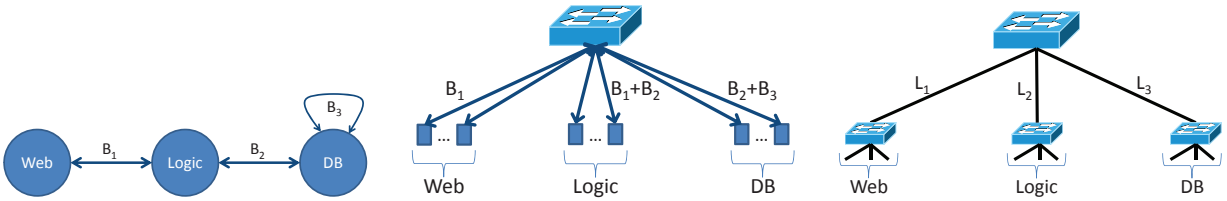
Let us consider two illustrative applications to highlight the shortcomings of prior models for abstracting and provisioning network bandwidth.

Many user-facing applications and sophisticated enterprise applications are composed of multiple tiers with complex traffic interactions [11, 12, 14]. Fig. 2(a) shows a simple example of a three-tiered web application with a frontend web tier, a business logic tier, and a backend database tier. Each tier contains multiple VMs and the edges of the communication graph are annotated with the bandwidth requirements between tiers. The second example is a real-

[1] In contrast, [18] observed only marginal increase (<5%) of completion times of various batch jobs when their per-VM bandwidths were capped at 33% below bottleneck-free capacity.

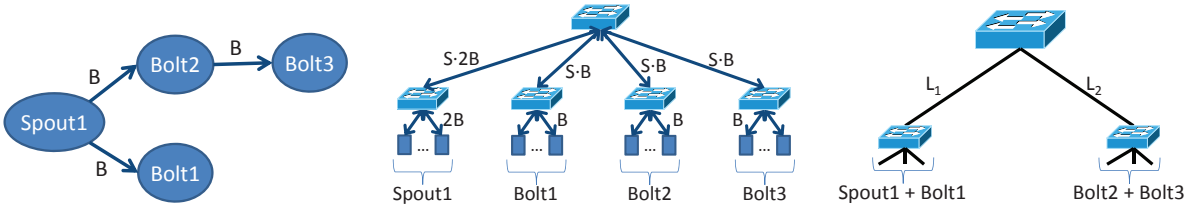
[2] CPU consumption: # of vCPUs × core speed × CPU busy %. BW/CPU likely understate BW usage; i) reported CPU% ranges [50,100] and ii) we treat it as 100% when not reported to make sure we do not over estimate the ratio. Redis [19] and VoltDB [20] benchmarks report transactions-per-second; we converted them to ranges of network throughput by assuming data size ranges [100,1500] bytes.

[3] At the server level, we compute the ratio of the server's NIC bandwidth and the aggregate server CPU cycles. At the Top-of-Rack (ToR) and aggregate switch levels, we compute the same ratio as the uplink bandwidth normalized by the total CPU cycles of servers under the ToR/aggregate switch.



(a) 3-tier web application (b) Hose modeling (c) Physical deployment example

Figure 2: Three tiered application example deployed using the hose model.



(a) Storm example (b) VOC Modeling (c) Physical deployment example

Figure 3: Storm [30] application example deployed using the VOC model.

time data analytics application shown in Fig. 3(a), implemented using Storm [30]. Storm is a popular platform for online machine learning, continuous computation on data streams, etc. Storm applications have two types of components, implemented using Java threads: "spouts", which are similar to mappers in MapReduce, and "bolts", which represent both a mapper and a reducer.

2.2 Shortcomings of Prior Models

The most commonly used abstractions are variants of the hose model and the pipe model. These are often a poor fit for modeling many applications, as we describe next.

• **Hose Model:** In the hose model abstraction [4,7,8], all VMs are connected to a central (virtual) switch by a dedicated link (hose) having a minimum bandwidth guarantee. We consider a generalized hose model [8] where each VM can have a heterogeneous bandwidth guarantee (unlike [4,28]) to better match application requirements. While the hose model well describes batch applications with homogeneous all-to-all communication patterns [18], it does not accurately express the requirements for applications composed of multiple tiers with complex traffic interactions. It can also be severely inefficient in terms of resource utilization. Consider the example of Fig. 2(a) and assume that B_1 represents the typical bandwidth demand between one VM of the web tier and one VM of the business logic tier. B_2 is the bandwidth demand between one VM of the business logic tier and one VM of the database tier, while B_3 is the bandwidth demand between two database VMs representing traffic for maintaining database consistency. For simplicity, we assume an equal number of VMs in each tier and equal bandwidth requirements in both directions of each edge while ignoring bandwidth requirements for Internet access.

Fig. 2(b) presents the hose model guarantees for the example in Fig. 2(a). The fundamental problem is that the hose model *aggregates* the demands for multiple different communications – e.g., logic-DB (B_2) and DB-DB (B_3) for a database VM – into one hose. This prevents the cloud operator from accurately computing the required bandwidth on physical links and leads to inefficient bandwidth consumption. Suppose that each application tier is deployed on a separate subtree of the physical network as shown in Fig. 2(c). To satisfy the hose model representation, the bandwidth that must be reserved on link L_3 for each database VM would be B_2+B_3.[4] The hose model hides the fact that B_3 is intended for the communication within the DB tier rather than for communication with other tiers. The tenant does not actually need the full guarantee of (B_2+B_3) indicated by the hose model on link L_3, thus wasting B_3 on L_3.

In addition to being inefficient, the hose model also *fails to guarantee the required bandwidth in case of congestion.* In the 3-tier example of Fig. 4, simplified from Fig. 2(a), suppose $B_1 = 500$, $B_2 = 100$ (in Mbps) and that all the tiers are placed under the same switch with each VM in a separate server. The hose guarantee for the business logic VM would be the sum of the requirements $500 + 100 = 600$ Mbps. Suppose the bottleneck bandwidth towards the business logic VM is also 600 Mbps. If the business logic VM momentarily receives an aggregate of 500 Mbps traffic from web VMs and also another 500 Mbps from DB VMs, the total 1000 Mbps traffic exceeds the available bandwidth and the hose guarantee (both 600 Mbps). Because the hose model aggregates the requirements for two different communications (from web and DB), the model is agnostic to the actual guarantees that the business logic VM needs for different sources. Existing solutions would partition the 600 Mbps hose guarantee by TCP-like max-min fairness [7] and yield 300:300 (Mbps), assuming equal number of sending VMs from each tier, but fail to provide the 500 Mbps guarantee for the communication with the web tier.

• **Virtual Oversubscribed Cluster (VOC):** This model proposed in [4,6] enhances the hose model by organizing VMs into clusters, each with a hose model guarantee. Clusters are then connected together with per-cluster hoses with capacity of $B \cdot S/O$, where B is the guarantee of each VM inside the cluster, S is the size of the cluster (number of VMs) and O is the oversubscription factor [4]. Again, to better suit applications, we consider a generalized VOC model that accommodates different guarantees, sizes and oversubscription factors for each cluster, unlike the more constrained homogeneous model in [4].

The VOC model is also not well suited to represent most applications. Consider the Storm example in Fig. 3(a), where for simplicity we assume that each component has the same number of VMs S, and the outgoing bandwidth of each VM to a communicating component is B. Even for this simple example it is non-trivial to derive a good VOC model representation from many possible representations. Fig. 3(b) presents one possible mapping where each application component is represented as a VOC cluster. The resulting model is a relaxed VOC model with no oversubscription of

[4]We assume here that $B_2 + B_3 < B_1 + B_1 + B_2$, and so the minimum that needs to be reserved on L_3 is B_2+B_3 rather than $2B_1$+B_2.

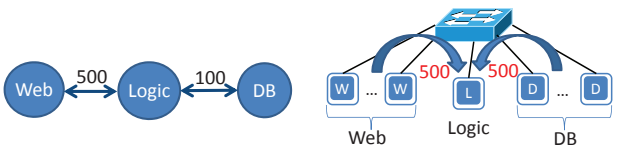


Figure 4: Hose fails to separately guarantee traffic to Logic from Web and DB in case of congestion.

the clusters. This model also fails to accurately capture the application's communication pattern as the Storm components do not communicate internally using that bandwidth.

The goal behind the VOC model is to isolate highly connected application tiers and place them in better connected topology subtrees. Having clusters that are not oversubscribed and that do not communicate among their VMs defeats the purpose of the VOC model, and, in fact, has an adverse effect. The placement algorithm may place the VMs of each Storm component in separate subtrees in an attempt to localize intra-component traffic, as Oktopus [4] does. This wastes bandwidth as the Storm threads communicate only between components.

Fig. 3(c) shows a potential deployment where two Storm components are placed in one branch of the physical tree while the other two are in a different branch. In this case, the bandwidth reservation on links L_1 and L_2 should be $S \cdot B$ given the communication pattern (since only "Spout1" communicates with "Bolt2" between the two branches). However, VOC will reserve twice this bandwidth since VOC is agnostic to actual inter-component communication pattern and requires $\min(3S \cdot B, 2S \cdot B) = 2S \cdot B$.

In essence, the VOC model also *aggregates* bandwidth requirements towards different components into a single hose and a single oversubscribed hose. This aggregation prevents: 1) determining the actual inter-component bandwidth needed at physical links, and 2) guaranteeing the required bandwidth in case of congestion (similar to Fig. 4). Recently, Hadrian [6] extended VOC by enabling each component X to list the components that X communicates with, e.g., Spout1: {Bolt1, Bolt2}, but this still aggregates requirements towards the listed components into a single oversubscribed hose and does not spell out the actual inter-component patterns.[5]

To see the significance of inter-component traffic, we analyzed the component-to-component traffic matrix from the `bing.com` datacenter, provided by the authors of [11]. In this example, we found that *most application components have high inter-component communications*. The inter-component traffic fraction of each component averages 91% over 500+ components. The total inter-component traffic constituted 65% of the entire datacenter traffic.Excluding traffic from management and common services, the *total* inter-component traffic becomes 37% of the entire non-management traffic; VOC model is still ineffective since the *per-component* ratio of inter-component traffic is still high at 85% on average.

Gatekeeper [9] tried to better model inter-component guarantees by allowing the composition of multiple hoses for each VM. In the example of Fig. 4, the business logic VM would be connected to two hoses: one for 500 Mbps guarantee towards the web tier and the other for 100 Mbps towards the DB tier. However, the hoses unnecessarily include intra-component traffic, wasting physical bandwidth to provide the unnecessary guarantees or failing to meet the intended guarantee: e.g., DB-DB traffic can hog the bandwidth intended for logic-DB traffic.

• **Pipe Model:** Another alternative is to specify bandwidth guarantees between pairs of VMs [3, 31, 32] as virtual pipes. While this model can precisely capture the application's traffic needs, it is too rigid and *it lacks statistical multiplexing*. Typically, the VMs belonging to different tiers that exchange data are selected by runtime load balancers, which do not guarantee perfectly uniform load distribution to every destination. Thus, load to each destination can vary over time even when the aggregate load is constant. Inability to update each pipe rapidly to tightly track their time-varying demand likely requires worst case reservations of peak load for each pipe [8]. For instance, the pipe model might force 2X overprovisioning based on a benchmark report for Amazon's Elastic Load Balancer that shows that the sum of the peak loads to each destination is at least double the peak aggregate traffic [33]. The pipe model is also tedious to use as the tenants can have hundreds and even thousands of VMs, leading to tens of thousands of pairwise guarantees, *hurting scalability* in placing tenant VMs (§5).

In summary, as opposed to the hose model and its variants that aggregate (or reduce) various communication patterns into a single hose guarantee, we need a new abstraction that spells out the actual communication pattern between pairs of components, similar to the pipe model, but without suffering from being tedious and slow like the pipe model.

3. TENANT APPLICATION GRAPH (TAG)

We propose the Tenant Application Graph (TAG), a new model that tenants can use to describe bandwidth requirements for applications. Unlike hose and VOC abstractions, which present models resembling physical networks, the TAG abstraction aims to model the actual communication patterns of applications. The TAG model leverages the tenant's knowledge of an application's structure to yield a concise yet flexible representation of the application's communication pattern.

A TAG model is *a graph, where each vertex represents an application component* (or *tier*, we use the two terms interchangeably to indicate the set of VMs performing the same function). Since most applications are conceptually composed of multiple tiers [12], a tenant can simply map each tier onto a TAG vertex. For example, for the application in Fig. 2(a) the tenant would identify three tiers: web, business logic, and database. One or more special components are used to model nodes external to the TAG tiers, e.g., the Internet, a storage service, another tenant, etc. Each component u has a 'size' attribute, N^u, denoting the number of VMs in that component, which is optional for the special components.

Tenants request bandwidth guarantees between tiers by placing directed edges between the corresponding vertices in the TAG model. Each directed edge $e = (u, v)$ from tier u to tier v is labeled with an ordered pair $< S_e, R_e >$ that represents per-VM bandwidth guarantees. Specifically, each VM in tier u is guaranteed bandwidth S_e for sending traffic to VMs in tier v, and each VM in tier v is guaranteed bandwidth R_e to receive traffic from VMs in tier u.

Having two values (sending and receiving) instead of a single bandwidth guarantee for each edge is useful when the sizes of the two tiers are different. If tiers u and v have sizes N^u and N^v, respectively, then the total bandwidth that TAG guarantees for traffic sent from tier u to tier v is $B_{u \to v} = \min(S_e \cdot N^u, R_e \cdot N^v)$ because the total outgoing traffic from tier u cannot exceed the total incoming traffic to tier v, and vice versa.

To model communication among VMs within tier u, the TAG model allows a self-loop edge of the form $e = (u, u)$ that is labeled with a single bandwidth guarantee SR_e, which represents both the sending and the receiving guarantee for each individual VM in that tier (or vertex). A self-loop edge is equivalent to a conventional hose model; each VM in tier u can be considered to be attached to

[5] In addition, Hadrian's extension to VOC is meant to model inter-tenant requirements instead of inter-component requirements.

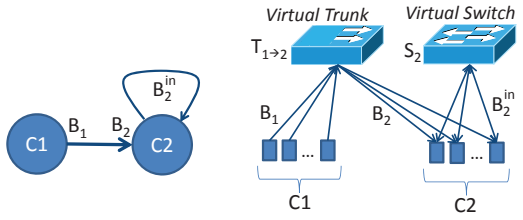


Figure 5: TAG model (a) example, (b) explained.

a virtual switch via a transmission hose of rate SR_e and a receive hose of rate SR_e.

Fig. 5(a) shows a TAG model for a simple example application with two tiers $C1$ and $C2$. In this example, a directed edge from $C1$ to $C2$ is labeled $< B_1, B_2 >$. Thus, each VM in $C1$ is guaranteed to be able to send at rate B_1 to the set of VMs in $C2$. Similarly, each VM in $C2$ is guaranteed to be able to receive at rate B_2 from the set of VMs in $C1$.

To better understand the TAG model, Fig. 5(b) shows an alternative way of visualizing the guarantees expressed in Fig. 5(a). The directional guarantees between C1 and C2 is represented by the *virtual trunk* $T_{1\to2}$. Each VM in $C1$ is connected to $T_{1\to2}$ by a dedicated directional link of capacity B_1; and $T_{1\to2}$ is connected through a directional link of capacity B_2 to each VM in $C2$. Thus, a virtual trunk can be seen as a *directional* hose model between the VMs of the two communicating tiers; $T_{1\to2}$ captures one-to-many/many-to-one send/receive bandwidth for each VM in $C1$/$C2$. This differs from two alternatives in modeling inter-tier guarantees: 1) guaranteeing the aggregate bandwidth between $C1$ and $C2$ through a tier-to-tier pipe, and 2) provisioning every send-receive VM pair with a pipe [31]. The alternatives lack the efficiency and flexibility benefits of using TAG that we will describe soon.

The TAG model example in Fig. 5(a) has a self-loop edge for tier $C2$, describing the bandwidth guarantees for traffic where both source and destination are in $C2$, e.g., the traffic between database servers in Fig. 2(a). A self-loop edge is equivalent to a hose model between the VMs of that tier. For example, in Fig. 5(b), each VM in $C2$ is connected through a bidirectional link of capacity B_2^{in} to a *virtual switch* S_2.

Note that hose and pipe models are special cases of TAG: a TAG with one component and a self-loop is the hose model, and a TAG with exactly one VM per component and no self-loops is the pipe model.

Benefits: Because the TAG abstraction mirrors real application structure, it is *intuitive, descriptive and easy to use*. Mirroring application structure enables the TAG model to be *efficient*, describing the bandwidth needs of complex structured applications accurately unlike hose or VOC models that often lead to over-allocation of bandwidth (§2.2).

Furthermore, the TAG model is *flexible*. Like the hose model, TAG presents per-VM guarantees, enabling it to take advantage of statistical multiplexing by specifying a guarantee based on the peak of the sum of VM-to-VM demands instead of the (typically larger) sum of peak demands needed by the pipe model. TAG is also flexible to dynamic re-sizing of tiers (known as *"auto-scaling"* [34]); per-VM bandwidth guarantees S_e and R_e typically do not need to change when tier sizes are changed by scaling. For example, Netflix's benchmark on AWS exhibited stable per-VM bandwidth while scaling up the number of VMs from 48 to 288 [24]. This is unlike a VM-to-VM pipe model [31, 32] where *per-pipe* bandwidth guarantees need to be recomputed whenever dynamic load-balancing or auto-scaling takes place, or else bandwidth must be heavily overprovisioned. This is also unlike guaranteeing an aggregate total bandwidth between components, which would have to change when components scale up/down.

Finally, grouping VMs by component in the TAG model enables tenants to specify a much smaller number of values than for the pipe model.[6]

Producing TAG Models: Users who understand the structure of their distributed applications can specify a matching TAG model and tune the component bandwidth guarantees to their needs. Alternatively, cloud orchestration systems like OpenStack Heat and AWS CloudFormation could be extended to generate TAG models automatically along with their current ability to automate application deployment and control application scaling. These systems use application templates, possibly provided as a library for users, that explicitly describe application components and their configuration and could be extended with bandwidth guarantee information for each component.

For users who do not know the structure and bandwidth demands of their applications, the provider or the user can try to infer an appropriate TAG model. We are exploring an approach that clusters VMs that exhibit similarity in their communication pattern, e.g., VMs that share a common set of destinations. For each VM, a feature vector is constructed based currently only on the VM-to-VM bandwidth weighted traffic matrix. The feature vector includes the VM's row and column entries, i.e., both outgoing and incoming traffic, and similarity is computed as the angular distance between vectors. A projection graph is formed containing one vertex for each VM and edges with weight set to the similarity between the VMs for the two incident vertices. Known algorithms [35, 36] that maximize graph modularity are applied to the projection graph to identify clusters of VMs with high edge weight within the cluster, indicating high similarity among the VMs. The TAG model is formed by treating each cluster as a component and using the traffic matrix bandwidths to identify all hose and trunk guarantees. When identifying these guarantees, we use a time series of traffic matrices to factor in savings from statistical multiplexing.

We applied this approach to the `bing.com` dataset to determine how well it could infer the known component structure given only the VM-to-VM traffic matrix. Using the metric of adjusted mutual information ranging from 0 (independent) to 1 (identical) clustering [37], we obtained on average 0.54 over 80 applications using Louvain clustering [35], indicating substantial commonality between the ground truth clustering and the inferred clusters, but also the need for further improvement.

Component-Level Graph Models: Graph models have been proposed to describe generic topologies between service components and desirable communication policies (security, priority, forwarding) between components, e.g., in Service-Oriented Architecture [38], Group-based Policy [31, 39] and Network Functions Virtualisation [40]. Our TAG model can be viewed as extending these models by explicitly representing *bandwidth guarantees* between communicating components.

4. DEPLOYING TAG

TAGs capture tenant application structures rather than physical network topologies. We present a VM placement algorithm that bridges the gap between the high level TAG to the low level physical infrastructure. As most cloud datacenter topologies are tree structures, we aim to efficiently deploy TAG instances on tree-

[6]To be even simpler for users, two edges in opposite directions between two tiers can be combined into a single undirected edge when the incoming/outgoing values for the tiers are the same (i.e., $S_{(u,v)} = R_{(v,u)}$ and $R_{(u,v)} = S_{(v,u)}$).

shaped physical topologies. For simplicity, we describe our algorithm assuming a single-rooted tree, however our algorithm can similarly be applied to a multi-rooted tree.

Existing network-aware approaches have focused on reducing bandwidth usage by localizing network traffic into a small region/sub-tree: thus colocating VMs that incur high network traffic between them [4, 11, 32]. In this section, we first derive the key conditions that enable bandwidth reduction through colocation. Next, we show that bandwidth reduction through colocation is often infeasible or undesirable due to high availability (HA) and other requirements. Finally, we present our VM placement algorithm that uses the bandwidth reduction conditions to efficiently deploy TAG models and also to improve high availability.

4.1 Bandwidth Required by TAG

To deploy an application that is described by a TAG model, the cloud provider must allocate sufficient bandwidth on physical links to support the bandwidth guarantees specified in the TAG model. For a given TAG model, we calculate the amount of bandwidth that must be allocated on the uplink of a particular subtree of a datacenter tree topology, when the subtree contains a subset of the TAG VMs. The subtree could be at any level of the topology, e.g., server, ToR switch, or aggregation switch; and the uplink connects the subtree to the rest of the datacenter topology.

Let X denote the set of TAG components placed in the subtree of interest, and $\overline{X}$ denote the set of all components that are outside this subtree. A component can be a member of both sets if it has some VMs in the subtree and the other VMs outside the subtree. Let N_X^t represent the number of VMs of component t that reside in the subtree, and let $N_{\overline{X}}^{t'}$ represent the number of VMs of component t' that reside outside the subtree.

For the TAG model, the bandwidth $C_{X,\text{out}}$ that must be allocated for outgoing transmission from X to $\overline{X}$ is given by summing up the requirement for each pair of components where the first component (t) has at least one VM in the subtree and the second component (t') has at least one VM outside the subtree. Define $B_{\text{snd}}^{t\to t'}$ as the bandwidth needed for each VM in component t for transmission to component t', and $B_{\text{rcv}}^{t\to t'}$ as the bandwidth needed for each VM in component t' for reception from component t. These values are obtained directly from the TAG model. Then, we have

$$\begin{aligned} C_{X,\text{out}} &= \sum_{t\in X}\sum_{t'\in\overline{X}} \min(N_X^t B_{\text{snd}}^{t\to t'}, N_{\overline{X}}^{t'} B_{\text{rcv}}^{t\to t'}) \\ &= B_{\text{trunk}} + B_{\text{hose}}. \\ B_{\text{trunk}} &= \sum_{t\in X}\sum_{\substack{t'\in\overline{X}\\ t'\neq t}} \min(N_X^t B_{\text{snd}}^{t\to t'}, N_{\overline{X}}^{t'} B_{\text{rcv}}^{t\to t'}). \qquad (1) \\ B_{\text{hose}} &= \sum_{t\in X\cap\overline{X}} \min(N_X^t B_{\text{snd}}^{t\to t}, N_{\overline{X}}^{t} B_{\text{rcv}}^{t\to t}). \end{aligned}$$

B_{trunk} is the inter-component requirement (i.e., virtual trunk) and B_{hose} is the intra-component requirement (i.e., hose).[7] The TAG bandwidth requirement in the opposite direction from $\overline{X}$ to X, say $C_{X,\text{in}}$, is calculated similarly.

4.2 Bandwidth Savings by Colocation

We derive key conditions that enable bandwidth savings through colocation from Eq. 1. Consider B_{hose} that defines the outgoing *hose bandwidth* needed at the subtree uplink. The min() term for B_{hose} defines each t's hose bandwidth across the subtree uplink. This term can be re-written as $\min(N_X^t, N^t - N_X^t)B_{\text{snd}}^{t\to t}$ where N^t is the total number of VMs of component t and because $B_{\text{snd}}^{t\to t} = B_{\text{rcv}}^{t\to t}$. As N_X^t increases from zero to N^t, the hose bandwidth increases from zero to $\frac{N^t}{2}B_{\text{snd}}^{t\to t}$ when $N_X^t = \frac{N^t}{2}$ and then decreases to zero. The hose bandwidth is zero either when all VMs of t reside in X or $\overline{X}$. From the standpoint of t in the subtree, hose bandwidth saving due to increasing degree of colocation happens only when

$$N_X^t > \frac{N^t}{2}. \qquad (2)$$

Thus, the necessary and sufficient condition to achieve hose bandwidth saving is that more than half the VMs of t must be colocated in the subtree.

The min() term for B_{trunk} in Eq. 1 defines the *virtual trunk bandwidth* from t to t' (where $t' \neq t$) across the subtree uplink, for outgoing traffic. Let $N^{t'}$ denote the total number of VMs of t'. Then the outgoing trunk bandwidth, denoted by $B1$, is computed as

$$B1 = \min(N_X^t B_{\text{snd}}^{t\to t'}, (N^{t'} - N_X^{t'})B_{\text{rcv}}^{t\to t'}). \qquad (3)$$

We next derive the worst-case trunk bandwidth requirement, $B2$, and compute the amount of trunk bandwidth saving as $B2 - B1$. The worst-case trunk bandwidth is incurred when all VMs of t' are placed outside of the subtree ($N_X^{t'} = 0$), in which case no bandwidth is saved and $B2 = \min(N_X^t B_{\text{snd}}^{t\to t'}, N^{t'} B_{\text{rcv}}^{t\to t'})$. For simplicity, we assume that the total sending rate from t is equal to the total receiving rate of t', i.e., $N^t B_{\text{snd}}^{t\to t'} = N^{t'} B_{\text{rcv}}^{t\to t'}$; we have $B2 = N_X^t B_{\text{snd}}^{t\to t'}$ since $N_X^t \leq N^t$. Then, the bandwidth that can be saved by (partial) colocation is:

$$B2 - B1 = \max(N_X^t B_{\text{snd}}^{t\to t'} - (N^{t'} - N_X^{t'})B_{\text{rcv}}^{t\to t'}, 0). \qquad (4)$$

The condition to have non-zero trunk bandwidth saving, $B2 - B1 > 0$, is given as

$$N_X^t B_{\text{snd}}^{t\to t'} + N_X^{t'} B_{\text{rcv}}^{t\to t'} > N^{t'} B_{\text{rcv}}^{t\to t'}. \qquad (5)$$

It is clear that the bandwidth saving increases as N_X^t and $N_X^{t'}$ increase, i.e., as more VMs of t and t' are colocated. The required uplink bandwidth is zero when hosting all VMs of t and t' in the subtree: $N_X^t = N^t$ and $N_X^{t'} = N^{t'}$. Again assuming that the total sending rate from t is equal to the total receiving rate of t', i.e., $N^t B_{\text{snd}}^{t\to t'} = N^{t'} B_{\text{rcv}}^{t\to t'}$, Eq. 5 becomes $N^{t'} N_X^t + N^t N_X^{t'} > N^t N^{t'}$; its *necessary* condition is easily proven to be:

$$N_X^t > \frac{N^t}{2} \text{ or } N_X^{t'} > \frac{N^{t'}}{2}. \qquad (6)$$

Thus, to achieve trunk bandwidth saving, more than half of the VMs of t or those of t' need to be colocated in the subtree, similar to the hose saving condition Eq. 2. Since Eq. 6 is a necessary but not sufficient condition for trunk savings, our placement algorithm verifies savings using Eq. 4 before colocating multiple tiers.

4.3 Disadvantages of Colocation

A naive approach to deploy a TAG would be to identify a set of TAG tiers that heavily communicate with each other and colocate them on the same subtree to save bandwidth, while ensuring valid placements by satisfying the requirement of Eq. 1. This is the approach taken by existing network-aware placement solutions [3, 4, 31] though they use different abstractions: hose, VOC, or pipe.

However, bandwidth saving by VM colocation can lead to single-point-of-failure and conflict with High Availability (HA) requirements. In addition, colocation can cause unbalanced utilization of the different types of resources available on each host (bandwidth, CPU, memory); in turn, this may increase the number of

[7] $C_{X,\text{out}}$ for the VOC model is similarly defined as $\min\left(\sum_{t\in X}\sum_{\forall t'\neq t} N_X^t B_{\text{snd}}^{t\to t'}, \sum_{t'\in\overline{X}}\sum_{\forall t\neq t'} N_{\overline{X}}^{t'} B_{\text{rcv}}^{t\to t'}\right) + B_{\text{hose}}$. From this, one can easily prove that the TAG model always requires a bandwidth allocation that is less or equal to that needed with the VOC model.

hosts and potentially the network bandwidth allocation needed to support a given application demand. We next elaborate on these two disadvantages of colocation.

Colocation vs. High Availability (HA). Colocation increases the chance for applications to experience downtime with a single server or switch failure. Thus, HA requirements may conflict with bandwidth-saving goals. An HA requirement is often expressed as anti-affinity (anti-colocation): the VMs of the same tier/service must reside on multiple servers or switches to retain service availability in case of a server/switch failure. Anti-affinity improves HA but depletes bandwidth saving from colocation. Anti-affinity also increases the need for bandwidth saving in oversubscribed datacenter networks: e.g., anti-affinity enforced at the tree level L_{AA} increases the chance for tenant traffic to consume the bandwidth of level $L_{\text{AA}}+1$ subtrees, which have smaller aggregate bandwidth than level L_{AA}.

Previous work addressing HA requirements for workload placement (for example [11, 41, 42]) has not, to our knowledge, also provided bandwidth guarantees. Some, notably Bodik et al. [11], have presented techniques to jointly maximize bandwidth savings and HA but without providing guarantees. Our placement algorithm provides both bandwidth guarantees as well as optional HA guarantees on a per-tenant basis. Even for tenants without HA requirements, we can opportunistically increase survivability when bandwidth is not the bottleneck.

Colocation vs. Efficient Resource Utilization. Colocation can reduce the efficiency of resource utilization when there are disproportionate demands for different resource types, e.g., network vs. CPU (Fig. 1). In Fig. 6, we consider an example of placing three hose components under a rack that has enough resources to accommodate the request. The placement in Fig. 6(c) localizes 16 Mbps bandwidth demands from components A and B under the two leftmost servers but cannot provide requested guarantees to component C. If there is no available VM slot outside this rack, this request will be rejected and this rack will be underutilized. Previous network-aware VM placement algorithms would blindly colocate A's VMs and B's VMs and yield this inefficient allocation. The alternative allocation in Fig. 6(d) does not save the total bandwidth at the server uplinks but efficiently utilizes both slot and bandwidth resources while providing the guarantees. The key to achieving this balanced and efficient utilization is *placing high-bandwidth and low-bandwidth demanding VMs together, even though they do not communicate with each other.* Note that this placement also improves HA.

Algorithm 1: VM scheduler (simplified)

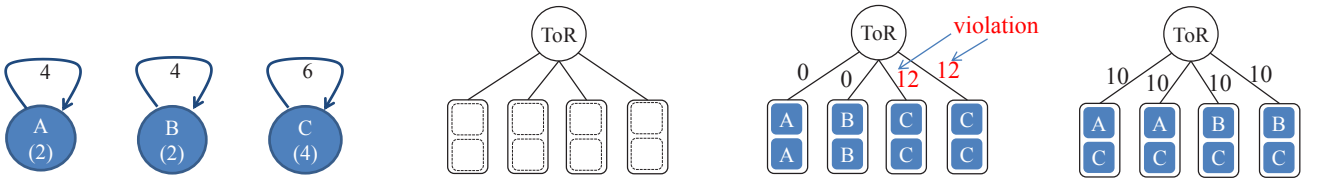
```
1  def AllocTenant(TAG g):
2      st = FindLowestSubtree(g, 0);
3      while st:
           /* Pass g's copy to Alloc.  map:  server
              locations of placed VMs                    */
4          map = Alloc(g, st);
5          if entire g in map:
               /* Reserve bandwidth for map upto root    */
6              ReserveBW(map, root); return map;
7          Dealloc(map, st);
8          if st is root: return False;
9          st = FindLowestSubtree(g, level(st)+1);
10     return False;
11 def Alloc(g, st):
12     map = {} ;
13     if level(st)== 0: /* st is a server.  reserve its
       slots and uplink BW.                               */
14         map[st] += g; ReserveBW(map,st);
15         return map;
       /* First, minimize BW usage by colocation         */
16     if BwSavingsFeasible():
17         map = Colocate(g, st);
18         g.Subtract(map);
       /* Second, balance resource utilization           */
19     if g.size > 0:
20         map += Balance(g, st);
21     if map:
22         if ReserveBW(map, st) is False:
23             Dealloc(map, st);
24             map = {};
25     return map;
26 def Colocate(g, st):
27     amap = {};
28     (g_sub, child) = FindTiersToColoc(g, st);
29     while g_sub.size > 0:
30         map = Alloc(g_sub, child);
31         amap += map; g.Subtract(map);
32         (g_sub, child) = FindTiersToColoc(g, st);
33     return amap;
34 def Balance(g, st):
35     amap = {};
36     (g_sub, child) = MdSubsetSum(g, st);
37     while g_sub.size >0:
38         map = Alloc(g_sub, child);
39         amap += map; g.Subtract(map);
40         (g_sub, child) = MdSubsetSum(g, st);
41     return amap;
```

4.4 VM Placement Algorithm

We aim to maximize the datacenter resource utilization by accepting as many TAG requests as possible while guaranteeing requested bandwidth. This problem is NP-hard similar to optimally placing hose models [8]. We first present our main heuristic in Algorithm 1, and extend it to support HA in §4.5. For brevity, we sketch the steps in the algorithm and define only major functions in the pseudocode. We assume identical VM types and slots; extending for heterogeneous cases is straightforward.

The algorithm starts with `AllocTenant`, which takes a TAG graph g as an input; g lists application components (vertices), communication edges, and their attributes (component size and bandwidth demand). In order to localize the traffic between the VMs in g, `FindLowestSubtree` in line **2** searches for a valid lowest-level subtree, `st`, under which the entire g is likely to fit. The search starts from the server level (0) and moves upward. g's demands for 1) VM slots and 2) bandwidth for communication with its external entities are validated against 1) the total number of VM slots available in the servers under `st` and 2) the available uplink bandwidth from `st` to the tree `root`.

`AllocTenant` in line **4** attempts to deploy g on `st` by calling `Alloc`, which reserves slot and bandwidth resources in the subtree below `st` and returns a `map` of allocated VMs and their server locations. If `map` covers all VMs of g, the algorithm successfully returns after reserving slot and bandwidth resources from `st` up to `root` and the cloud provider will launch VMs on the specified servers in `map`. `Alloc` may fail to allocate the entire g under `st`, because the bandwidth availability of the links below `st` is not guaranteed by `FindLowestSubtree`; the actual requirements on those links are unknown until we know the exact number of VMs of each tier that will be placed in each server under `st`. In case of such failure, `Dealloc` in line **7** releases the resources reserved for the VMs in `map`. The algorithm next tries to allocate g under a new subtree at `st`'s parent level (line **9**). When moving to the parent level of `st`, the siblings of `st` will be considered for placement of g either in its entirety or split across multiple siblings. The algorithm continues



(a) Request for 3 hose components, total 8 VMs and 40 Mbps demand.

(b) Rack with 4 servers, each with 2 slots and 10 Mbps NIC.

(c) Colocation for saving bandwidth of A and B.

(d) No saving but efficient resource utilizations.

Figure 6: Colocation for bandwidth saving can hurt efficient resource utilization.

to move to higher levels of the tree until either placement succeeds or else placement finally fails at the tree `root` (line **8**).

`Alloc` (line **11**) is a recursive function, invoked by its two subroutines: `Colocate` and `Balance`. (We describe `Alloc` and the subroutines in a depth-first manner.) Before reaching the subroutines, `Alloc` first checks if the target `st` is a server: if so, it allocates `g`'s VMs on the server and returns; we assume sufficient bandwidth is always available for communication between VMs in the same server. If `st` is a ToR or higher level switch, `Alloc` strategically allocates `g`'s VMs over `st`'s child nodes by `Colocate` and `Balance`.

`Colocate` is invoked only when bandwidth saving through colocation is feasible (line **16**), determined by the size conditions (Eqs. 2 and 6) and the HA requirement (§4.5). As the algorithm recurses down the tree by subsequent invocations of `Alloc`, the number of VMs of a hose tier (or a pair of trunk tiers) in `g` may become less than 50% of the original tier size(s); bandwidth saving thus becomes infeasible. Likewise, if anti-affinity is required at level L_{AA} with 50% or higher worst case survivability (R_{WCS}, defined in §4.5), it is impossible to place >50% of a tier under a subtree at level-L_{AA} or lower.

`FindTiersToColoc` in `Colocate` identifies a set of VMs $\mathtt{g}_{\mathtt{sub}}$ ($\subset$ `g`) that provide (hose, trunk or both) bandwidth saving through colocation by using the size conditions of Eqs. 2-6. It excludes from $\mathtt{g}_{\mathtt{sub}}$ low-bandwidth tiers (e.g., A and B in Fig. 6) that can subsequently be placed together with high bandwidth tiers (e.g., C in Fig. 6) to increase resource utilization across different resource types, e.g., network and CPU. The idea is to identify tiers having low per-VM bandwidth demand compared to the per-slot available bandwidth of `st`'s child nodes, and then place VMs of these low-bandwidth tiers together with VMs of higher bandwidth tiers, where the VMs of each higher bandwidth tier are not themselves colocated (due to the size or HA constraints tested on algorithm line **16**).[8] Note that this strategy places low-bandwidth VMs and high-bandwidth VMs together even when the two types of VMs do not communicate with each other.

To achieve this balance of slot and bandwidth utilization, `Balance` is invoked (line **20**) to deploy the VMs, remaining in `g` after `Colocate`, for which bandwidth saving is infeasible. `MdSubsetSum` in `Balance` tries to find the best `child` node of `st` and the best set of VMs $\mathtt{g}_{\mathtt{sub}}$ ($\subset$`g`) that will lead both slot and uplink utilization of `child` to approach 100%; this is similar to a known Multi-Dimensional Subset-Sum problem. We extended the standard one-dimensional greedy algorithm [43] to three dimensions (slot, in BW, out BW) by using the utilization ratio of each resource as a common metric. We also improved its speed by iterating over `g`'s tiers, instead of every VM, as the VMs in the same tier have the same demands.

Algorithm Complexity: We compare asymptotic complexity of Algorithm 1 with that of Oktopus and two pipe-based algorithms [3, 32]. All four algorithms recurse over the hierarchical datacenter structure; we analyze the complexity of each recursion. We fix the datacenter tree degrees as constant. The complexity for `Alloc` to find a valid placement of a TAG graph `g` in each recursion is $O(T^2)$, where T is the number of tiers in a tenant, because `FindTiersToColoc` iterates over the tier-to-tier edges in `g` to find tiers with large bandwidth saving. Oktopus's complexity at each recursion is $O(K)$; K is a mean tier size. But Oktopus deploys each tier of a VOC separately and every tier invokes a separate top-level recursion; per-recursion complexity becomes $O(KT)$. We found CloudMirror and Oktopus runtimes are comparable, their difference within an order of magnitude: CloudMirror is faster for some tenants and vice versa. The work in [32] performs a min-cut over a VM-to-VM graph at each recursion with $O(N^4)$ complexity; N is the number of VMs of a tenant ($N = KT$). SecondNet [3] reduced the complexity to $O(N^3)$ at the cost of optimality but it is still >1000 times slower than CloudMirror or Oktopus when $K \cong 10$, $T \cong 5$ (from the `bing` dataset excluding the management services).

4.5 Providing High-Availability

We next extend the main VM placement algorithm to support two approaches in meeting HA goals in addition to providing bandwidth guarantees: 1) guaranteeing anti-affinity for scenarios with strict HA requirements and 2) opportunistic anti-affinity for scenarios that do not have strict HA requirements but would still benefit from increasing HA.

Guaranteeing Anti-Affinity. Guaranteeing network bandwidth for predictable performance and achieving high availability are both first class goals for many applications. We found cloud providers tend to deploy fault-resilient networking [44], especially at core switches, but there is no such mechanism for server failures. Thus, anti-affinity – placing VMs of a tier under multiple subtrees – is more desired at server level to increase worst-case survivability (WCS) of the tier. WCS is defined as the smallest fraction of VMs of the same tier that remain functional during a failure of a single subtree at level-L_{AA} in a datacenter [11].[9] In this paper we use servers as default fault isolation domains and set default L_{AA} to server level. To guarantee a required WCS, R_{WCS}, we extend the main algorithm and set an upper bound for N^t_X, the number of VMs placed under a subtree of L_{AA} or lower level:

$$N^t_X <= \max(1, \text{int}(N^t * (1 - R_{\text{WCS}}))), \tag{7}$$

Our evaluations will show that guaranteeing WCS may decrease datacenter utilization while increasing fault tolerance.

Opportunistic Anti-Affinity. For tenants not paying for HA

[8] This could be a poor strategy if subsequently arriving tiers/tenants require large bandwidth allocations. To obtain needed bandwidth, the algorithm would have to reverse its earlier decisions and choose locality maximization for previously placed tiers, a capability we currently do not consider for simplicity.

[9] Ref. [11] modeled fault-domain also considering powerline failures, which our algorithm can be extended to incorporate.

guarantees, we still opportunistically improve HA by distributing their VMs across servers when bandwidth saving is infeasible (from the size conditions of Eqs. 2-6) or *undesirable*, a new property we now introduce to capture relative difference between bandwidth availability and bandwidth demand. For example, bandwidth saving could be less desirable at server level because server uplink bandwidth can support the demand of most applications (Fig. 1). Formally, we determine bandwidth saving *desirability* by comparing the available bandwidth averaged over unallocated slots under `st` against the average per-VM bandwidth demand of input `g`, factoring in the expected contributions of future tenant VMs (predicted based on previous arrivals). If the former is smaller than the latter, bandwidth saving is deemed desirable.

To implement the opportunistic approach, we make three modifications to the main algorithm. We modify line **16** to consider also the desirability such that `Colocate` is invoked only when it is both *feasible* and *desirable*. Similarly, `FindLowestSubtree` starts searching from the lowest level where bandwidth saving is desirable, instead of blindly starting from the server level, to facilitate VM placements across multiple servers. The third modification goes to `MdSubsetSum`, which originally returned as many VMs as possible in g_{sub} (line **36**) to fill in the chosen `child` node (in a resource-balanced way) and leaved more sibling child nodes with more slots available for future tenants to do colocation. We modify `MdSubsetSum` to return only one VM in g_{sub} and to select the best `child` for that VM when bandwidth saving is *not* desirable. This encourages distributed VM allocations over **all** child nodes while still achieving balanced slot and bandwidth utilization of the child nodes. Our evaluation will show that this opportunistic approach can greatly improve average WCS at the cost of marginally decreased datacenter utilization while preserving all bandwidth guarantees. The decrease is due to the imperfect estimation of future demands and thus suboptimal desirability decision.

5. EVALUATION

We next evaluate the benefit of our TAG model and the performance of our proposed CloudMirror placement algorithm (CM). We evaluate 1) their efficiency in (a) *reserving less network bandwidth* and (b) *accepting more tenant requests* compared to other models and placement algorithms. Our evaluation isolates the separate impacts of the TAG model and the CM placement algorithm on these efficiency metrics. We also evaluate 2) the ability of our placement algorithm to *guarantee and improve availability*, while providing bandwidth guarantees. Finally, we verify 3) the feasibility of deploying our solution on a real testbed.

To evaluate 1) and 2), we wrote a simulator in Python that implements CM to deploy TAG models, and Oktopus [4] to deploy virtual cluster (VC) – hose with homogeneous bandwidth – and VOC models. We found VC always performed worse than VOC and TAG. Thus, we omit the VC results and use VOC as a baseline. We substantially improved the Oktopus algorithm [4] by: handling the case when "Alloc" fails to allocate the requested slots, placing clusters of the same VOC under a common subtree to localize inter-cluster traffic, and relaxing the VOC model to allow arbitrary sizes, cluster bandwidth and core bandwidth for different clusters.

Simulation Setup: We simulate a tree-shaped 3-level network topology inspired by a real cloud datacenter, with 2048 servers. For simplicity, we assume all VMs have identical CPU and memory requirements, and each server can host 25 VMs.

Each simulation run consists of 10,000 Poisson tenant arrivals and departures. Arriving tenants are uniformly sampled at random from a pool of 80 tenants, described below. We vary the mean arrival rate (λ) to control the *load* on a datacenter while keeping tenant dwell time (T_d) fixed; the load is $\frac{T_s \lambda T_d}{2048 \times 25}$, where the mean tenant size (#VMs) is T_s and the denominator is the total VM slots.

Algorithm	Server	ToR	Agg
CM+TAG	3209.0	1006.8	0.7
CM+VOC	3266.5 (1.02)	1230.1 (1.22)	1.7 (2.55)
OVOC	2978.8 (0.93)	1299.7 (1.29)	14.7 (22.08)

Table 1: Reserved bandwidth (Gbps) for `bing` workload. Values in () report bandwidth ratio compared to CM+TAG.

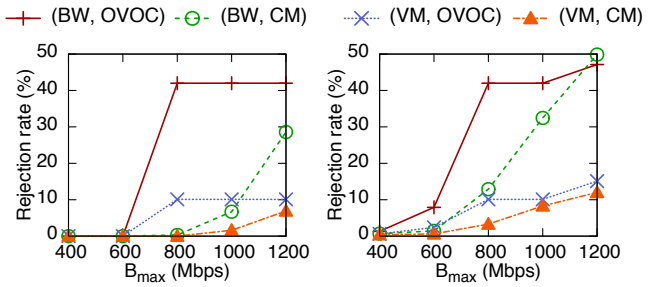
Each tenant arrival has a single TAG or VOC request. Experiments draw arrivals from one of three workloads: empirical datasets from `bing.com` [11] and `hpcloud.com` [29], and a synthetic workload. Due to space constraints, we primarily present results from the `bing` workload. The `bing` workload consists of a set of services, the service size ranging from one to a few hundred VMs. The services constitute a diverse range of job types (interactive web services or batch data-processing) and communication patterns (e.g., linear, star, ring, mesh; shown in Fig. 7 in [11]), and some have large intra-service demands (similar to MapReduce). A set of central management and logging services communicates with every other service, but mostly at low bandwidth. We remove the common management/logging services and their traffic, as similar to [11], to create a set of isolated tenants which our experiments randomly sample to simulate arrivals. In total there are 80 tenants in the pool: their mean size T_s is 57, with some large tenants over 200 VMs in size; the largest tenant has 732 VMs. We consider each service as corresponding to a component/tier in the TAG model and to a cluster in the VOC model.

5.1 Simulation Results

We first evaluate the main CloudMirror algorithm (CM) that does not consider HA.

Bandwidth Reserved: Table 1 lists the aggregate bandwidth reserved on uplinks from the server, ToR, and agg switch network levels, for three combinations of model and placement algorithm – CM+TAG, CM+VOC and Oktopus+VOC (OVOC). Since CM is not designed to place VOC models, CM+VOC uses the placement obtained by CM+TAG but reports the bandwidth allocation resulting from modeling the tenants using VOC. For fair comparison, we simulate an ideal network topology with unlimited network capacity; all three combinations deploy the same set of tenants. We simulate only tenant arrivals (no departure) and measure the aggregate bandwidth usage of the deployed tenants when the first tenant rejection happens due to lack of VM slots.

Table 1 shows results for the `bing.com` workload containing both intra- and inter-component traffic. Comparing VOC and TAG models with the same CM placement, VOC consumes more bandwidth than TAG at all three network levels, because the VOC model lacks inter-component traffic pattern information and, thus, reserves unnecessary bandwidth (§2.2). However, the bandwidth advantage of TAG over VOC is small at server level. This is because the services (components) in the `bing` workload are either smaller or $> 2\times$ larger than the server size (25 VM slots), leaving no trunk bandwidth savings advantage for TAG through colocation (Eq. 6) at server level. Trunk bandwidth saving is possible at ToR and agg levels resulting in a greater difference between TAG and VOC models. Comparing placement algorithms, CM+VOC consumes more server-level bandwidth than Oktopus+VOC (OVOC) because CM avoids colocation for low-bandwidth components, instead spreading their VMs across servers to achieve balanced resource utilization (as in Fig. 6(d)); we demonstrate the benefits of resource balancing below where we introduce bandwidth capac-



(a) Load = 50% (b) Load = 90%

Figure 7: Rejection rates with various bandwidth scaling. Legend format: (metric, algorithm).

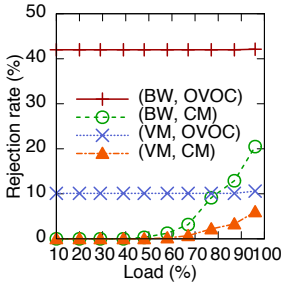


Figure 8: Rejection rates with varying loads. B_{max} = 800Mbps.

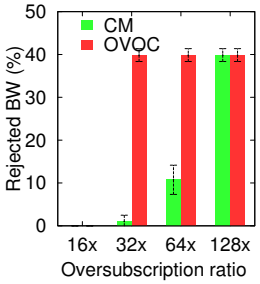


Figure 9: Bandwidth rejection rate across different oversubscription ratios.

ity constraints into the network topology. At ToR and agg switch levels, CM+VOC allocates less bandwidth than OVOC. CM placement can achieve savings through its strategy of colocating components with high inter-component traffic in the same subtree, unlike Oktopus which places components independently.

Experiments (not shown) using a synthetic workload, formed by artificially mixing different application sizes and types (e.g., three tier web services and MapReduce jobs) and experiments using the `hpcloud` workload yielded results similar to Table 1.

Workload Rejection Rate: We extend the experiment to a topology with constrained bandwidth capacity and we simulate tenant arrivals and departures while varying the mean arrival rate. Now some tenant requests may be rejected due to lack of network capacity, and thus we compare the rate of rejected tenant requests. We assume each server has a 10G uplink to its ToR switch; and ToR-aggregation and aggregation-core links are oversubscribed by a 32:8:1 ratio, mimicking real datacenters [2].

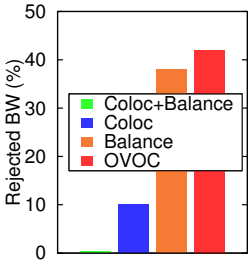
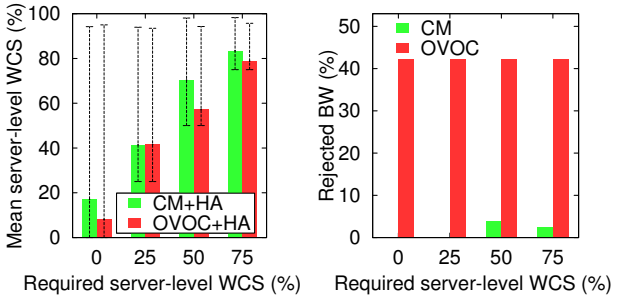


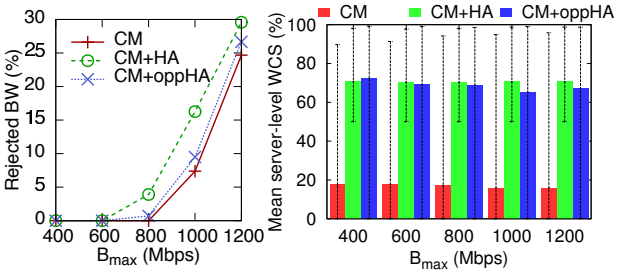
Figure 10: Micro-benchmarking of CM.

Fig. 7 plots the ratios of rejected tenants' #VMs and aggregate bandwidth relative to those of the total tenant arrivals. Rejection of a tenant can happen due to insufficient available bandwidth and/or CPU/memory resources (slots). The bandwidth values in the `bing.com` workload dataset are relative, not absolute. We scale the bandwidth values such that the average per-VM demand (B_{vm}) of the tenant with the largest B_{vm} becomes the target per-VM bandwidth (B_{max}). Fig. 7(a) shows that for some B_{max}, CM (=CM+TAG) can deploy almost all requests while OVOC rejects up to 40% of bandwidth requests. Fig. 7(b) also shows a substantial difference between CM and OVOC for a different target load. In



(a) Achieved WCS (b) Rejected BW

Figure 11: Impact of guaranteeing WCS. The error bar denotes max and min achieved WCS.



(a) Rejected BW (b) Achieved WCS

Figure 12: Comparison of different HA mechanisms. CM: default approach w/o any HA. CM+HA: guarantee 50% WCS. CM+oppHA: opportunistic HA.

all experiments, tenant rejection rate is less than 2.2%, but rejected tenants tend to have unusually large VM/bandwidth requirements.

The results with varying load are shown in Fig. 8. OVOC fails to deploy a set of tenants having large slot or bandwidth demands even at low loads while CM efficiently places most of them. We stress-test CM and OVOC by increasing the network topology oversubscription ratio. Fig. 9 illustrates that CM is resilient to highly bandwidth-constrained network environments while OVOC is quickly incapable of deploying tenants.

To evaluate the impact of two subroutines of the CM algorithm – `Coloc` and `Balance` – we deactivate each subroutine one by one and plot the TAG bandwidth rejection rate of each case (`Coloc` only and `Balance` only) with that of the original CM (`Coloc+Balance`) and, just for reference, OVOC in Fig. 10. Colocation is clearly the main factor in accepting more resource requests but `Balance` also contributes by preventing a subtree from leaving its compute resource heavily underutilized while its network is maxed out and vice versa (Fig. 6). Even without `Coloc`, which seeks colocation benefits between components, the `Balance`-only approach performed close to OVOC in Fig. 10.

High Availability (HA): Some tenants need guaranteed HA as much as guaranteed bandwidth. Our HA extension, CM+HA, can place VMs while guaranteeing bandwidth and worst-case survivability (WCS) at a specified tree level, L_{AA}. For comparison, we also extended Oktopus to implement WCS-guarantees. Fig. 11 shows that the required WCS (R_{WCS}, x-axis) is achieved with both algorithms when L_{AA}=SERVER. CM+HA achieves higher average WCS across the deployed application components than OVOC because CM tries a balanced resource utilization by colocating VMs with opposite resource requirements (high bandwidth VM and low bandwidth VM) instead of blindly colocating VMs from the same component. Guaranteeing HA with a higher WCS requirement

increases the bandwidth rejection rate. The increase is small in Fig 11(b) because bandwidth is not a bottleneck at the server level; we observed higher increases when we set L_{AA} to higher-levels.

We next evaluate the opportunistic anti-affinity enhancement to CM described in §4.5. Fig. 12 shows that the opportunistic HA (CM+oppHA) can achieve high average WCS comparable with the guaranteed HA approach, while yielding bandwidth rejection rates as low as the default CM algorithm (see the points where $B_{max} = 1000$ in Fig. 12(a)). With its opportunistic non-guaranteed approach, CM+oppHA may achieve high or low WCS values close to zero (see error bars in Fig. 12(b)). We also tested when L_{AA}=TOR and observed very similar patterns with Fig. 12, except that CM+HA rejected more BW than in Fig. 12(a)).

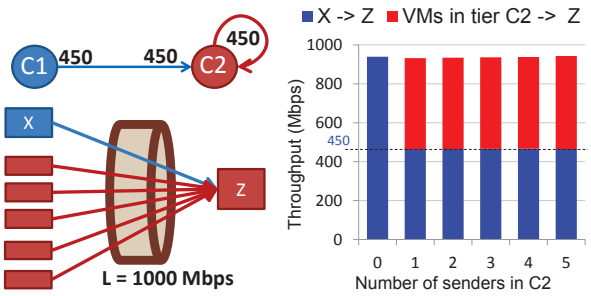
Algorithm runtime: CM, implemented in Python (single-thread), typically runs within 200 msec for tenants of up to 100s of VMs and up to a few seconds for tenants of up to 1000 VMs, demonstrating its scalability. Our CM and Oktopus implementations have similar runtimes. We ran the simulations on Standard-type compute instances in HP Public Cloud.

We also implemented the SecondNet [3] algorithm that places pipe models, using C++ for core libraries. We converted the `bing` TAG models to idealized pipe models by dividing each hose and trunk guarantee uniformly across the corresponding pipes (instead of planning realistically for worst-case as described in §2.2). SecondNet is faster than another pipe model algorithm [32] but still slow to deploy the `bing` workload especially at high datacenter utilization, taking tens of minutes to place one large tenant. Assuming ideal placement, idealized pipe models are fundamentally more bandwidth-efficient than the more flexible TAG models. However, SecondNet produced less efficient resource allocations than CM+TAG leading to higher tenant rejection and also much longer execution time. This demonstrates the sheer complexity of placing VM-VM pipes rendering the pipe model less practical. Since pipe is a special case of TAG, we were able to evaluate running CM to deploy the idealized `bing` pipe models, and observed CM+pipe consuming 8% less bandwidth than SecondNet. CM+pipe runtime, while slower than CM+TAG, was comparable to that of SecondNet; note that CM is implemented in Python while SecondNet in C++ and Python.

5.2 Guarantee Enforcement Prototype

Enforcement of TAG model guarantees can be implemented as a straightforward patch to most prior solutions for enforcing hose-model bandwidth guarantees, such as [4, 5, 7, 9]. The intuition is that all these frameworks rate-limit pairs of source-destination VMs (the rate-limit is based on the two VMs' bandwidth guarantees, their current communication pattern, the degree of congestion, etc.). Since the TAG model can be seen as being composed from a set of (directional) hose models, the only conceptual change to be made to these frameworks is to identify to which hose a particular source-destination VM pair belongs. We have implemented a proof of concept based on this approach in ElasticSwitch, a recent proposal for enforcing work-conserving hose-model bandwidth guarantees [7]. This patch consists of 30 lines of code.

Fig. 13(a) shows a simple experiment scenario and topology. We aim to show that the two bandwidth guarantees of VM Z from tier $C2$ are isolated from each other (one guarantee for traffic from $C1$ and one for $C2$ intra-tier traffic). For simplicity we set $B_1 = B_2 = B_2^{in} = 450$ Mbps; the bottleneck link towards VM Z is 1 Gbps and we leave 10% of the bandwidth unreserved. In our experiment, VM Z receives TCP traffic from VMs in both tiers $C1$ and $C2$. We use a single sender, VM X, in tier $C1$, and we vary the number of senders in tier $C2$. Fig. 13(b) plots application-level



(a) Experiment setup (b) TCP throughput of VM Z

Figure 13: TAG guarantees using ElasticSwitch.

throughput of VM Z between the senders in tiers $C1$ and $C2$ as we increase the number of senders in tier $C2$. Traffic from VM X of tier $C1$ is protected from the larger intra-tier traffic.

6. DISCUSSION AND FUTURE WORK

Here we briefly discuss several extensions of the TAG model and the overall CloudMirror system design.

Large-scale variations in load will trigger tenants to scale up or down by "auto-scaling", which is flexibly handled by the TAG model. We plan to extend our placement algorithm to better support auto-scaling. Smaller-scale load variations, which do not trigger scaling, can vary bandwidth requirements over time; CloudMirror can adopt existing approaches, such as workload profiling [18] or history-based prediction [45], to be even more efficient.

We believe that the TAG model is versatile and could potentially express other requirements and policies besides bandwidth, including latency, security, access control, reliability, and auto-scaling. Generalizing the TAG concept with additional application level requirements is a potential topic for follow-on work.

Automatic generation of TAG requirements is another ripe area of research. We have sketched a measurement-based system to automatically identify application components and their bandwidth requirements from raw VM-VM level traffic. We plan to conduct a rigorous evaluation of the outlined techniques.

Cloud operators often provide popular application components as infrastructure services (e.g., block storage, front-end web cluster). Tenants can use TAG to express their bandwidth demands between those infrastructure service components and the other components they bring in. When every tenant component is an infrastructure service (e.g., Platform-as-a-Service), TAG can be used internally by the cloud provider to capture the bandwidth demands between the components.

We expect that TAG and CM placement principles can be applied to future datacenters and workloads. For example, next-generation resource-disaggregated datacenters [46, 47] will likely interconnect pools of compute capacity with pools of non-volatile memory (NVRAM) in a hierarchical network. The NVRAM will have throughput and access times orders of magnitude faster than the rotating storage media it displaces, thus imposing much higher bandwidth demands on future datacenter-wide interconnect. This challenge further strengthens the need for efficient and balanced resource allocation, as provided by CloudMirror. We envision extending the TAG model to capture bandwidth guarantees between compute elements and NVRAM; each TAG component currently defining a set of similar VMs can be split into one component that defines a set of similar compute elements, and one component that defines a set of NVRAM, with virtual trunks added to specify bandwidth guarantees between these components.

7. CONCLUSION

Migrating mission critical applications to cloud environments demands a network virtualization solution with performance guarantees. We introduced an efficient network virtualization solution, *CloudMirror*, with three components – a network abstraction called Tenant Application Graph (TAG) that represents the true application communication pattern, an efficient VM placement strategy that efficiently utilizes datacenter resources while providing high availability, and a runtime mechanism that enforces the application bandwidth requirements. Our simulation experiments using real application traces demonstrate significantly improved datacenter resource efficiency over existing solutions.

Acknowledgments: We thank Peter Bodik and the other authors of [11] for providing us with the data we used in the presented experiments. We also thank the anonymous reviewers and our shepherd, Chuanxiong Guo, for their guidance on the paper.

8. REFERENCES

[1] "Facebook Future-Proofs Data Center With Revamped Network." http://tinyurl.com/6v5pswv.
[2] N. Farrington and A. Andreyev, "Facebook's Data Center Network Architecture," IEEE Optical Interconnects, 2013.
[3] 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," ACM CoNEXT, 2010.
[4] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards Predictable Datacenter Networks," ACM SIGCOMM, 2011.
[5] V. Jeyakumar, M. Alizadeh, , D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical Network Performance Isolation at the Edge," USENIX NSDI, 2013.
[6] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea, "Chatty Tenants and the Cloud Network Sharing Problem," USENIX NSDI 2013.
[7] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing," ACM SIGCOMM, 2013.
[8] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A Flexible Model for Resource Management in Virtual Private Networks," ACM SIGCOMM, 1999.
[9] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes, "Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks," USENIX WIOV, 2011.
[10] J. Lee, M. Lee, L. Popa, Y. Turner, S. Banerjee, P. Sharma, and B. Stephenson, "CloudMirror: Application-Aware Bandwidth Reservations in the Cloud," USENIX HotCloud, 2013.
[11] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving Failures in Bandwidth-Constrained Datacenters," ACM SIGCOMM, 2012.
[12] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani, "Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud," ACM SIGCOMM, 2010.
[13] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of The ACM*, vol. 56, Feb 2013.
[14] "AWS Architecture Center." http://aws.amazon.com/architecture/.
[15] "Amazon - Every 100ms delay costs 1% of sales." http://tinyurl.com/k635quz.
[16] V. Jeyakumar, A. Kabbani, J. C. Mogul, and A. Vahdat, "Flexible Network Bandwidth and Latency Provisioning in the Datacenter," in *arXiv:1405.0631*, 2014.
[17] "WikiBench: Web hosting benchmark." http://www.wikibench.eu/.
[18] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The Only Constant is Change: Incorporating Time-varying Network Reservations in Data Centers," ACM SIGCOMM, 2012.
[19] "Redis Benchmark & Rackspace Performance VMs." http://tinyurl.com/omyzyax.
[20] "877,000 TPS with Erlang and VoltDB." http://tinyurl.com/naoakwl.
[21] "Testing Vyatta 6.5 R1 under VMware." http://tinyurl.com/l7u9gfa.
[22] J.-C. Huang, M. Monchiero, Y. Turner, and H.-H. Lee, "Ally: OS-transparent packet inspection using sequestered cores," ACM/IEEE ANCS, 2011.
[23] J. Summers, T. Brecht, D. Eager, and B. Wong, "Methodologies for Generating HTTP Streaming Video Workloads to Evaluate Web Server Performance," ACM SYSTOR, 2012.
[24] Netflix Tech Blog, "Benchmarking Cassandra Scalability on AWS." http://tinyurl.com/3ogo79c.
[25] H. Li and A. Michael, "Intel Motherboard Hardware v2.0," tech. rep., Open Compute Project.
[26] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, J. Rexford, R. Wattenhofer, and M. Zhang, "Dionysus: Dynamic Scheduling of Network Updates," ACM SIGCOMM, 2014.
[27] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic Engineering with Forward Fault Correction," ACM SIGCOMM, 2014.
[28] "Rackspace: Welcome to Performance Cloud Servers; Have Some Benchmarks!." http://tinyurl.com/pam57uy.
[29] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware task placement for cloud applications," ACM IMC, 2013.
[30] "Storm: Distributed and fault-tolerant realtime computation." http://storm-project.net/.
[31] T. Benson, A. Akella, A. Shaikh, and S. Sahu, "CloudNaaS: A Cloud Networking Platform for Enterprise Applications," ACM SOCC, 2011.
[32] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," IEEE INFOCOM, 2010.
[33] Brian Adler, "Load Balancing in the Cloud: Tools, Tips, and Techniques." RightScale Technical Whitepaper.
[34] "Amazon Web Services - Auto Scaling." http://aws.amazon.com/autoscaling/.
[35] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech.*, 2008.
[36] M. Rosvall and C. T. Bergstrom, "Maps of random walks on complex networks reveal community structure," *PNAS 2008*.
[37] N. X. Vinh, J. Epps, and J. Bailey, "Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance," *Journal of Machine Learning Research*, vol. 11, pp. 2837–54, 2010.
[38] A. Klein, F. Ishikawa, and S. Honiden, "Towards Network-aware Service Composition in the Cloud," ACM WWW, 2012.
[39] "OpenDaylight, Group-based Policy project." http://tinyurl.com/n42h3ju.
[40] "ETSI, Network Functions Virtualisation." http://tinyurl.com/maaqnng.
[41] F. Hermenier, J. Lawall, and G. Muller, "BtrPlace: A Flexible Consolidation Manager for Highly Available Applications," *IEEE TDSC*, vol. 10, no. 5, 2013.
[42] E. Bin, O. Biran, O. Boni, E. Hadad, E. K. Kolodner, Y. Moatti, and D. H. Lorenz, "Guaranteeing high availability goals for virtual machine placement," ICDCS, 2011.
[43] B. Przydatek, "A fast approximation algorithm for the subset-sum problem," 1999.
[44] "Cisco Virtual Switching Systems (VSS)." http://tinyurl.com/yqg97w.
[45] K. LaCurts, J. Mogul, H. Balakrishnan, and Y. Turner, "Cicada: Introducing Predictive Guarantees for Cloud Networks," USENIX HotCloud, 2014.
[46] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network Support for Resource Disaggregation in Next-Generation Datacenters," ACM HotNets, 2013.
[47] K. Asanovic, "FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers," USENIX FAST 2014 Keynote.