ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing

Lucian Popa Praveen Yalagandula* Sujata Banerjee Jeffrey C. Mogul†
Yoshio Turner Jose Renato Santos
HP Labs, Palo Alto, CA

ABSTRACT

While cloud computing providers offer guaranteed allocations for resources such as CPU and memory, they do not offer any guarantees for network resources. The lack of network guarantees prevents tenants from predicting lower bounds on the performance of their applications. The research community has recognized this limitation but, unfortunately, prior solutions have significant limitations: either they are inefficient, because they are not work-conserving, or they are impractical, because they require expensive switch support or congestion-free network cores.

In this paper, we propose ElasticSwitch, an efficient and practical approach for providing bandwidth guarantees. ElasticSwitch is efficient because it utilizes the spare bandwidth from unreserved capacity or underutilized reservations. ElasticSwitch is practical because it can be fully implemented in hypervisors, without requiring a specific topology or any support from switches. Because hypervisors operate mostly independently, there is no need for complex coordination between them or with a central controller. Our experiments, with a prototype implementation on a 100-server testbed, demonstrate that ElasticSwitch provides bandwidth guarantees and is work-conserving, even in challenging situations.

Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General

Keywords: Cloud Computing, Bandwidth Guarantees, Work-Conserving

1. INTRODUCTION

Today, cloud networks are shared between tenants in a best-effort manner. For this reason, current cloud providers cannot offer any guarantees on the network bandwidth that each virtual machine (VM) can use. The lack of bandwidth guarantees prevents tenants from predicting lower bounds on the performance of running their applications in the cloud, and from bounding the cost of running these applications, given the current pricing models [3]. Further, the lack of bandwidth guarantees impedes the transfer of enterprise applications to public clouds; many enterprise applications require predictable performance guarantees, but cloud network performance has been shown to vary [4] and congestion does occur in datacenters [6, 24]. For these reasons, some cloud customers may be willing to pay extra for bandwidth guarantees in the cloud.

Bandwidth guarantees can be achieved through static reservations [4, 11, 21]. In this way, cloud providers can offer tenants an experience similar to their own enterprise facilities, at a lower cost. However, static reservations lead to inefficient utilization of the network capacity, as the share of a tenant A cannot be used by another tenant B when A is not fully utilizing it, *i.e.,* static reservations are not *work-conserving*. Considering that the traffic in datacenters is bursty in nature and that the average utilization is low [6, 24], by multiplexing networking resources in a work-conserving manner, cloud providers can offer tenants a significantly better experience than static reservations, since tenant jobs would complete faster. At the same time, providers would improve the utilization of their own infrastructure.

Thus, we aim to design a cloud datacenter network that:

- *Provides Minimum Bandwidth Guarantees:* Each VM is guaranteed a minimum absolute bandwidth for sending/receiving traffic.
- *Is Work Conserving:* If a link L is the bottleneck link for a given flow, then L should be fully utilized.
- *Is Practical:* The solution should be implementable and deployable today (*i.e.,* work with commodity unmodified switches and existing network topologies) and scale to large cloud datacenters.

Existing proposals for sharing cloud networks, *e.g.,* [4, 11–13, 18, 20, 22], do not achieve all of the above goals simultaneously. For example, Oktopus [4] and SecondNet [11] are not work conserving and FairCloud [18] requires expensive support from switches, not available in today's commodity hardware (see §2 and §8 for more details on related work).

In this paper, we propose ElasticSwitch, a solution that achieves our goals. ElasticSwitch is fully implementable inside hypervisors, and does not require any support from switches. Tenants request minimum bandwidth guarantees by using the hose model [4, 9, 12, 18, 20] shown in Fig. 1. The hose model offers the abstraction that all VMs of one tenant appear to be connected to a single virtual switch through dedicated links. ElasticSwitch could also be used with other abstractions based on the hose model, such as the TAG model [14] or a hierarchical hose model, similar to one in [4] (§9).

ElasticSwitch decouples its solutions for providing bandwidth guarantees and for achieving work-conservation into two layers, which both run in each hypervisor. The higher layer, called *guarantee partitioning* (GP), ensures that the hose-model guarantee for each VM is respected, regardless of the network communication pattern. The guarantee partitioning layer divides the hose-model bandwidth guarantee of each VM X into pairwise VM-to-VM guarantees between X and the other VMs with which X communicates. The lower layer, *rate allocation* (RA), achieves work-conservation by dynamically increasing rates beyond the guarantees allocated by guarantee partitioning, when there is no congestion. For this purpose, rate allocation employs a TCP-like mechanism to utilize all available bandwidth between pairs of VMs. The

*Currently at Avi Networks
†Currently at Google

SIGCOMM'13, August 12–16, 2013, Hong Kong, China.

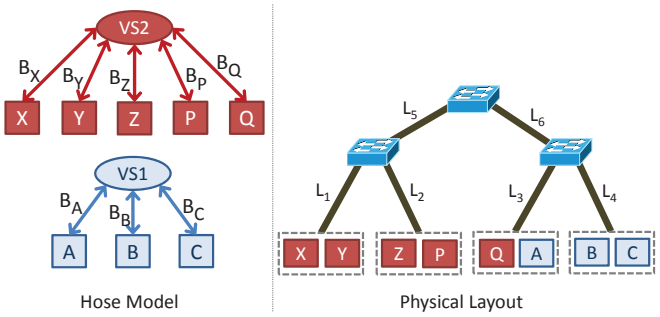


Figure 1: Example hose-model bandwidth guarantees for two tenants (Blue and Red) and the mapping for their virtual networks to a physical network. In the hose model, each VM V has a dedicated link of capacity B_V to a non-blocking virtual switch.

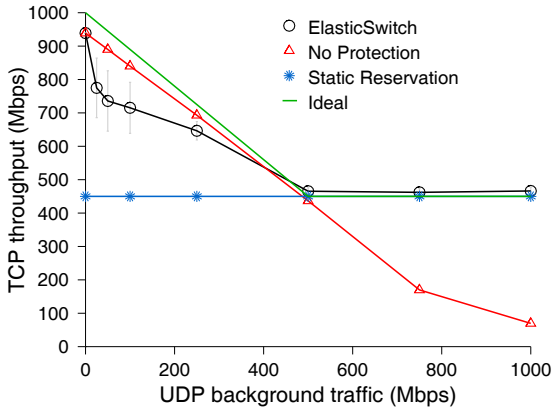


Figure 2: Bandwidth of one VM-to-VM TCP flow guaranteed 450Mbps, with varying UDP background rates

rate allocation layer dynamically borrows unallocated capacity, or capacity guaranteed to other VMs but not currently being used.

ElasticSwitch is a *distributed solution*, where hypervisors operate without the use of any complex coordination in addition to the normal flow of traffic. This is unlike prior hypervisor-based proposals for providing bandwidth guarantees, such as Oktopus [4] and SecondNet [11], which require frequent communication with a central controller and thus have limited scalability.

Fig. 2 shows an example of how ElasticSwitch gives minimum-bandwidth guarantees, unlike a "no protection" system, but is work-conserving, unlike a static-reservation system. In this example, one TCP flow, with a guarantee of 450Mbps, competes with UDP background traffic at various rates, sharing a 1Gbps link. When there is spare capacity, ElasticSwitch yields lower TCP throughput than the no-protection system; this gap is a function of parameters we can set to trade off between accurate bandwidth guarantees and full utilization of spare capacity (§7). The ideal work-conserving behavior is also plotted.

Contributions: In this paper, we describe:

1. A hypervisor-based framework that enables work-conserving bandwidth guarantees without switch modifications (§3). We construct this framework in two layers: a guarantee partitioning layer providing guarantees, and a rate allocation layer that provides work conservation, by grabbing additional bandwidth when there is no congestion.
2. Algorithms for guarantee partitioning (§4) and rate allocation (§5), based on hypervisor-to-hypervisor communication, which ensure that each VM's hose-model guarantee is respected, regardless of the other VMs in the network, and that the network is efficiently utilized.
3. A prototype implementation of ElasticSwitch (§6), and an evaluation in a 100-server datacenter network testbed (§7).

2. PROBLEM AND BACKGROUND

Our goal is to design a cloud datacenter network that provides minimum bandwidth guarantees for VMs communicating inside the cloud, in an efficient, scalable, and easily deployable way.

Bandwidth guarantees: Infrastructure-as-a-Service (IaaS) providers (either public or private) need to ensure performance isolation among all tenants sharing the underlying physical infrastructure. However, most academic and industrial work on virtualization has focused on compute and storage, and has only recently addressed network performance virtualization. Even today, many public cloud providers, such as Amazon EC2, offer no network guarantees, and thus tenants experience highly variable network bandwidth (by a factor of five in some cases [4]).

To model bandwidth guarantees, we focus on the *Hose* model, a well-understood model which mimics practical switched networks [4,9,12,18,20]. For example, in Fig. 1 the Blue tenant sees a virtual switch *VS1*, to which each of its VMs is connected via a dedicated link with a specified "hose" bandwidth. For simplicity, we use symmetric hoses, with equal ingress and egress bandwidths. However, it is easy to extend ElasticSwitch to use asymmetric hoses.

Given a tenant and the hose model for its virtual network, cloud datacenters need an *admission control* mechanism to determine if that tenant's network can be deployed on the physical network without violating the guarantees for the existing tenants. This problem has been studied in the past. For example, Oktopus [4] determines the placement of VMs in the physical network such that the physical link capacities ($L1$-$L6$ in the example figure) can support the hose model guarantees for all tenants. In the example of Fig. 1, assuming the hose-model bandwidths for both tenants are 500Mbps, and all link capacities are 1Gbps, the physical layout shown in the figure could *potentially* satisfy the guarantees.

However, just placing VMs without any enforcement of bandwidth limits does not *necessarily* satisfy the hose model guarantees. For example, in Fig. 1, Red VMs X and Y can send traffic to VM Q at 1Gbps, interfering with the traffic from Blue VM B to VM A.

The main issue addressed by this paper is the enforcement of bandwidth guarantees under hose models. We assume that the admission control and VM placement are existing components in the cloud computing infrastructure.

Efficiency: While it would be possible to enforce bandwidth guarantees through strict static provisioning of each link among the tenants, it would be inefficient in terms of network utilization. Consider the example in Fig. 1. Suppose both tenants are running MapReduce jobs. When Red VM Q is not using any bandwidth (e.g., the Red tenant is in the Map phase), Blue VM A could blast at 1Gbps and thus potentially shorten its shuffle phase by half. So in this example, and in general, a *work-conserving* enforcement policy could substantially improve the performance of a large fraction of applications. Since network link bandwidths are often significantly larger than guarantees (*e.g.,* 10Gbps vs 100Mbps), and since

Table 1: Summary of previous approaches and comparison to ElasticSwitch

Related Work	BW Guarantee Model	Work-Conserving	Control Model	Switch Requirements	Topology Requirements
Oktopus [4]	Hose, Virtual Oversubscribed Cluster	No	Centralized	None	None
SecondNet [11]	Hose, VM-to-VM	No	Centralized	MPLS	None
Gatekeeper [20]	Hose	Yes	Distributed	None	Congestion-Free Core
EyeQ [12]				ECN	
Seawall [22], NetShare [13] FairCloud [18] (PS-L/N)	No Guarantees (source/ tenant fair sharing)	Yes	Distributed	None	None
FairCloud [18] (PS-P)	Hose	Yes	Distributed	per VM Queues	Tree
ElasticSwitch	**Hose**	**Yes**	**Distributed**	**None**	**None**

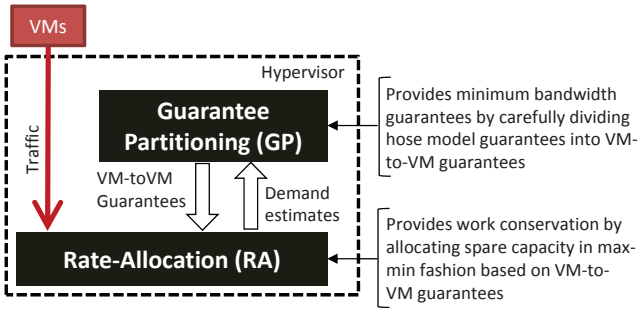


Figure 3: Overview of ElasticSwitch

datacenter traffic is bursty [6, 24], work-conservation can give an order of magnitude more bandwidth to a VM.

Thus, given a choice, customers would prefer providers offering work-conserving bandwidth guarantees, compared to providers offering only one of these two properties.

Scalability: Large public cloud providers host several thousands of tenants, each with tens to thousands of VMs. Further, traffic demands change rapidly and new flows arrive frequently, *e.g.,* a datacenter can experience over 10 million flows per second [6].

Given the high frequencies at which demands change and flows arrive, the provider must also adjust rate limits at a high frequency. Using a centralized controller to adjust these limits would scale poorly, since each VM can compete for bandwidth over an arbitrary set of congested links, and the controller would have to coordinate across all VMs and all links in the network. This would entail tremendous computation costs, and communication costs for control traffic. For example, Tavakoli *et al.* [25] estimate that a network of 100K servers needs 667 NOX controllers just for handling the flow setups, which is a significantly simpler problem. Furthermore, making the controller sufficiently available and reliable would be extremely expensive.

Thus, our goal is to design a *distributed* bandwidth guarantee enforcement mechanism that can scale to large cloud datacenters.

Deployability: We aim for our solution to be deployable in any current or future datacenter. Towards this end, we design our solution with three requirements. First, we want our solution to work with commodity switches, and to not assume any non-standard features. This reduces the equipment costs for cloud providers, who are quite cost-sensitive.

Second, we do not want our solution to depend on the network topology. For example, it should work on topologies with different over-subscription factors. Most existing networks are oversubscribed. While oversubscription factors have decreased over the last decade (from 100:1 to 10:1), oversubscription has not disappeared. All of the datacenters studied in [6] were oversubscribed; commercial datacenters had an oversubscription ratio of 20:1. Given that average network utilization is low, cost concerns may encourage oversubscription for the foreseeable future. We are aware of tens of cloud datacenters under construction that are significantly oversubscribed.

Third, to be applicable in a wide variety of cloud datacenters, our solution should be agnostic to applications and workloads.

Recent work: Recently, researchers have proposed different solutions for sharing cloud datacenter networks, but none of them simultaneously meets all of the above goals. Table 1 summarizes these solutions; we discuss them in detail in §8.

3. ElasticSwitch OVERVIEW

In this paper, we focus on a single data center implementing the Infrastructure as a Service (IaaS) cloud computing model. For simplicity of exposition, we discuss ElasticSwitch in the context of a tree-based physical network, such as a traditional data center network architecture or the more modern multi-path architectures like fat-tree [1] or VL2 [10]. For multi-path architectures, we assume the traffic is load balanced across the multiple paths by an orthogonal solution, *e.g.,* [2, 15, 19]. (ElasticSwitch can be applied to other topologies, see §9).

ElasticSwitch decouples providing bandwidth guarantees from achieving work conservation. Thus, ElasticSwitch consists of two layers, both running inside each hypervisor as shown in Fig. 3. The first layer, *guarantee partitioning* (GP), enforces bandwidth guarantees, while the second layer, *rate allocation* (RA), achieves work conservation.

The GP layer provides hose model guarantees by transforming them into a set of absolute minimum bandwidth guarantees for each source-destination *pair of VMs*. More specifically, the GP component for a VM X divides X's hose model guarantee into guarantees to/from each other VM that X communicates with. The guarantee between source VM X and destination VM Y, $B^{X\to Y}$, is computed as the minimum between the guarantees assigned by X and Y to the $X \to Y$ traffic.

The GP layer feeds the computed minimum bandwidth guarantees to the RA layer. Between every pair of VMs X and Y, the RA layer on the host of VM X uses a rate-limiter to limit the traffic. The rate-limit assigned by RA does not drop below the guarantee $B^{X\to Y}$, but can be higher. Specifically, RA aims to grab available bandwidth in addition to the provided guarantee, when the path from X to Y is not congested. RA shares the additional bandwidth available on a link L in proportion to the bandwidth guarantees of the source-destination VM pairs communicating on L. For example, assume $B^{X\to Y}$ = 200Mbps and $B^{Z\to T}$ = 100Mbps (X, Y, Z and T are VMs). In this case, sharing a link L larger than 300Mbps only between $X\to Y$ and $Z\to T$ is done in a 2:1 ratio; *e.g.,* if L is 1Gbps, $X\to Y$ should get 666Mbps and $Z\to T$ 333Mbps. For this purpose, rate allocation uses a mechanism similar to a weighted version of a congestion control algo-

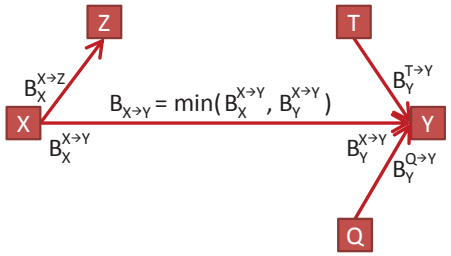


Figure 4: Example of guarantee partitioning

rithm, such as TCP, where each VM-to-VM communication uses a single weighted flow (TCP flows have equal weights). RA is similar to Seawall [22] and the large body of work on weighted TCP, *e.g.,* MulTCP [8], TCP-LASD [16].

Each hypervisor executes GP and RA periodically for each hosted VM. GP is performed less frequently than RA, since the guarantees allocated for the VM-to-VM flows of a VM X should be updated only when demands change to/from X, while rates allocated for those VM-to-VM flows should be updated when demands change on any congested link used by X.

VM placement: ElasticSwitch is orthogonal to the VM placement strategy, as long as the admission control criterion is satisfied, *i.e.,* the sum of the bandwidth guarantees traversing any link L is smaller than L's capacity.

External traffic: Traffic to/from hosts located outside the cloud must also be considered when providing bandwidth guarantees inside the cloud, since external and internal traffic typically share the same datacenter network links. For brevity, we describe only one way to model the external traffic. In this model, all external hosts appear as a single node attached to the virtual switch in the hose model; the guarantee for the aggregate external traffic only applies up to the egress from the datacenter. For this purpose, the external traffic must be routed through proxies executing ElasticSwitch.

4. GUARANTEE PARTITIONING (GP)

The distributed GP layer partitions the hose-model guarantee of a VM X into VM-to-VM guarantees. GP aims to achieve two goals: (1) *safety*, meaning that the hose-model guarantees of other VMs in the network cannot be violated by the assigned VM-to-VM guarantees and (2) *efficiency*, *i.e.,* VM X's throughput is not limited below what X would obtain if it were communicating through a physical network with capacity equivalent to X's hose-model guarantee.

We partition guarantees because implementing the hose model requires rate enforcement at the granularity of VM-to-VM pairs; per-VM limiters do not suffice. For example, depending on the communication pattern and demands, some of the VM-to-VM flows outgoing from VM X can be bottlenecked in the hose model at the source X, while other VM-to-VM flows from X can be bottlenecked in the hose model at the destination; this situation requires VM-to-VM limiters to emulate the hose model.

For a VM X, suppose $\mathbb{Q}_X^s$ denotes the set of VMs that are sending traffic to X and $\mathbb{Q}_X^r$ those receiving from X (almost always $\mathbb{Q}_X^s = \mathbb{Q}_X^r$). For any VM Y in $\mathbb{Q}_X^r$, X's hypervisor assigns a bandwidth guarantee $B_X^{X\to Y}$ for the communication to Y. Independently, for each VM $Y \in \mathbb{Q}_X^s$ X's hypervisor assigns guarantee $B_X^{Y\to X}$ for the traffic from Y. The total of such guarantees sums up to X's hose-model guarantee, B_X, in each direction, *i.e.,* $\sum_{Y\in\mathbb{Q}_X^r} B_X^{X\to Y} = \sum_{Y\in\mathbb{Q}_X^s} B_X^{Y\to X} = B_X$. (Note that it is straightforward to adapt ElasticSwitch to use an asymmetric hose model, with different incoming and outgoing guarantees.)

As we illustrate in Fig. 4, the key to ElasticSwitch's ability to provide hose model guarantees is to set the pairwise guarantee between source X and destination Y to the minimum of the guarantees allocated by X and Y:

$$\mathbf{B^{X\to Y} = \min(B_X^{X\to Y}, B_Y^{X\to Y})}$$

This allocation ensures the safety property for GP because, on any link L, the sum of the bandwidth guarantees for the VM-to-VM flows does not exceed the link bandwidth allocated by admission control for the hose model on link L. This is easy to see, since X's guarantee is divided between its flows, and each VM-to-VM flow gets the minimum of the guarantees allocated by source and destination.[1] For instance, for X's incoming traffic, $\sum_{V\in\mathbb{Q}_X^s} B^{V\to X} = \sum_{V\in\mathbb{Q}_X^s} \min(B_X^{V\to X}, B_V^{V\to X}) \le \sum_{V\in\mathbb{Q}_X^s} B_X^{V\to X} = B_X$.

Hence, the guarantees assigned by GP do not oversubscribe the reserved guarantees on any link, and, if we were to rate-limit all the VM-to-VM flows of VM X to these values, the guarantees of any other VM in the network are not affected by the traffic to/from X.

For example, to implement the Blue tenant's hose model from Fig. 1 in the physical topology (shown in the same figure), the Blue tenant should be given a minimum guarantee for its incoming/outgoing bandwidth on link L_3. Assuming each server hosts at most two VMs, the Blue tenant competes on L_3 with, at most, the bandwidth of the flows communicating with VM Q. However, the total bandwidth guarantee of the VM-to-VM flows of the Red tenant on link L_3 will always be less or equal to B_Q.[2] Thus, by ensuring that B_A+B_Q is less than the capacity of L_3, GP ensures that no tenant can violate the hose models of other tenants.

The process of assigning guarantees for VM-to-VM flows is applied for each VM, regardless of whether that VM is a source or destination. However, to compute the guarantee $B^{X\to Y}$ between VMs X and Y, the hypervisor of the source VM X must know the guarantee allocated by Y's hypervisor to $X\to Y$, *i.e.,* $B_Y^{X\to Y}$. ElasticSwitch does this with special *remote-guarantee control packets* sent by destination hypervisors to source hypervisors. Initially, each flow is given the guarantee allocated by the source hypervisor.

One remaining question is how the hypervisor of VM X divides X's guarantee between its VM-to-VM flows. The naive solution is to simply divide X's guarantee equally between its VM-to-VM flows. This approach works well when all VM guarantees in the hose model are equal, and all the traffic demands are either unsatisfied or equal. However, this is typically not the case in practice. For example, many of the flows are short (send a few packets) or have inherently limited demands.

We first describe GP in a static setting, where VM-to-VM flows have constant traffic demands and new flows are not generated, and then describe GP in a dynamic environment.

GP in a static context: In ElasticSwitch, we divide a VM's guarantee between its VM-to-VM flows in a *max-min* fashion based on traffic demands. For example, assume B_X=100Mbps for a VM X. Also assume that X communicates with three other VMs Y, Z and T, and the demand to Y is only 20Mbps, while demands to Z and T are very large (unbounded). In this case, guarantee partitioning assigns a guarantee of 20Mbps to the $X\to Y$ path (*i.e.,* $B_X^{X\to Y}$ = 20Mbps), and a guarantee of 40Mbps to each of $X\to Z$ and $X\to T$.

GP in a dynamic context: In practice, actual traffic demands are unknown and vary in time. ElasticSwitch estimates demands of

[1]We often refer to a guarantee as assigned by VM X, although guarantees are always assigned by the hypervisor hosting X.

[2]We assume here that $B_Q < B_X + B_Y + B_Z + B_P$, since the hose-model reservation on link L_3 is $\min(B_Q, B_X + B_Y + B_Z + B_P)$.

VM-to-VM flows by measuring their throughputs between epochs (information in fact provided by the RA layer). If the demand estimate is smaller (by a significant error margin) than the allocated guarantee, the VM-to-VM flow is considered bounded and its guarantee can be reduced. Otherwise, the flow was not satisfied and its guarantee should be increased, if possible. We describe this process from the perspective of a new flow.

When a new VM-to-VM flow starts, its demand is unknown. To optimize for the bi-modal distribution of flows typically observed in datacenters, in which most flows are very small and a few are quite large [6, 24], we set its demand estimate (and guarantee) at a small value. If, at the next periodic application of GP, we detect that the flow did not utilize its guarantee, then we deem the flow as bounded and use its measured demand in the previous epoch as the estimate (guarantee) for the next epoch (plus an error margin). If the flow did utilize its guarantee in the previous epoch, then we exponentially increase its guarantee. We increase the guarantee only up to the fair share of unbounded VM-to-VM flows to/from the VM for which GP is applied.

We emphasize that the above discussion refers only to new *VM-to-VM* flows. However, most of the TCP flows that a VM X initiates will be to VMs that X has already communicated with in the recent past (*e.g.,* [7] reports that each service typically communicates with a small number of other services). When all TCP flows between two VMs are short, the guarantee for that VM-to-VM flow will simply have a small value (since the demand will be small).

If all flows of one VM have bounded demands, any unallocated part of the bandwidth guarantee of that VM is redistributed (proportional to demands), such that the VM's entire guarantee is allocated. This allows a timely response if the demand of a flow increases.

Weak interdependence for efficiency: Up to now, we described GP as applied for each VM X independently of other VMs. However, to achieve efficiency and ensure X can fully utilize its guarantee in the hose model, it is useful to take into account the guarantees assigned by the remote VM with which X communicates. Specifically, since for the communication between $X \rightarrow Y$ only the minimum of $B_X^{X\rightarrow Y}$ and $B_Y^{X\rightarrow Y}$ is used, some of X's or Y's guarantees might be "wasted." Such potential waste can only occur for senders; intuitively, receivers simply try to match the incoming demands, so they cannot "waste" guarantees.

For example, assume that in Fig. 4 all demands are infinite and all VMs are part of the same hose model with identical guarantees, B. In this case, the guarantee assigned to the flow $X \rightarrow Y$ at Y's hypervisor, $B_Y^{X\rightarrow Y}$, will be $\frac{B}{3}$, which is smaller than the guarantee assigned by X's hypervisor, $B_Y^{X\rightarrow Y} = \frac{B}{2}$. Thus, a part of X's guarantee could be wasted because the $X \rightarrow Y$ flow is bottlenecked at Y. Thus, we should assign $B_X^{X\rightarrow Z}$ to be $\frac{2B}{3}$ instead of $\frac{B}{2}$.

To address this situation, senders take into account receivers' assigned guarantees. Specifically, if destination Y allocates a smaller guarantee than sender X (*i.e.,* $B_X^{X\rightarrow Y} > B_Y^{X\rightarrow Y}$), and if Y marks this flow as unbounded[3], meaning that Y allocates to $X \rightarrow Y$ its maximum fair share, the sender simply deems the demand of the flow as the destination's guarantee and hence sets $B_X^{X\rightarrow Y} = B_Y^{X\rightarrow Y}$ for the next epoch. Thus, our approach ensures that taking the minimum of source and destination VM-to-VM guarantees does not limit VMs from achieving their hose model guarantees. Note that Y must mark the flow as unbounded since, otherwise, Y's lower guarantee for the flow means that either: (a) the $X \rightarrow Y$ flow indeed has lower demand than the guarantees assigned by either X or Y, or (b) Y misclassified the flow, a situation that should be resolved at the next iteration.

Because GP is not entirely independent between VMs, a change in one VM-to-VM flow's demand or a new flow can "cascade" to the guarantees of other VMs. This cascading is loop-free, necessary to fully utilize guarantees, and typically very short in hop-length. For example, if all tenant VMs have equal bandwidth guarantees, updates in guarantees cascade only towards VMs with strictly fewer VM-to-VM flows. More specifically, a new VM-to-VM flow $X_1 \rightarrow X_2$ indirectly affects guarantees on the cascading chain X_3, X_4, X_5 iff: (i) all VMs fully utilize guarantees; (ii) flows $X_3 \rightarrow X_2$, $X_3 \rightarrow X_4$, and $X_5 \rightarrow X_4$ exist; and (iii) the $X_3 \rightarrow X_2$ guarantee is bottlenecked at X_2, $X_3 \rightarrow X_4$ at X_3, $X_5 \rightarrow X_4$ at X_4.

Convergence: The guarantee partitioning algorithm converges to a set of stable guarantees for stationary demands. We do not present a formal proof, but the intuition is as follows. Assume X is (one of) the VM(s) with the lowest fair-share guarantee for the unbounded incoming or outgoing VM-to-VM flows in the converged allocation; *e.g.,* if all VMs have the same hose bandwidth guarantee and all demands are unsatisfied, X is the VM that communicates with the largest number of other VMs. Then, X will (i) converge in the first iteration and (ii) never change its guarantee allocation. The senders (or receivers) for X will use the guarantees allocated for X and not change them afterwards. Thus, we can subtract X and its flows, and apply the same reasoning for the rest of the VMs. Hence, the worst case convergence time is is on the order of the number of VMs.

However, we do not expect the convergence of GP to be an issue for practical purposes. In fact, we expect convergence to occur within the first one/two iterations almost all the time, since multi-step convergence requires very specific communication patterns and demands, described earlier for the cascading effect.

In dynamic settings, convergence is undefined, and we aim to preserve safety and limit transient inefficiencies. GP's use of the minimum guarantee between source and destination ensures safety. We limit transient inefficiencies by not considering new flows as unbounded, and by applying GP frequently and on each new VM-to-VM flow. We evaluate transient inefficiencies in Section 7.

5. RATE ALLOCATION (RA)

The RA layer uses VM-to-VM rate-limiters, such that: (1) the guarantees computed by the GP layer are enforced, and (2) the entire available network capacity is utilized when some guarantees are unused; in this case, excess capacity is shared in proportion to the active guarantees.

We control rate-limiters using a weighted TCP-like rate-adaptation algorithm, where the weight of the flow between VMs X and Y is $B^{X\rightarrow Y}$, the bandwidth provided by the GP layer. We compute a shadow rate as would result from communicating using a weighted TCP-like protocol between X and Y. When this rate is higher than the minimum guarantee $B^{X\rightarrow Y}$, we use it instead of $B^{X\rightarrow Y}$, since this indicates there is free bandwidth in the network.

Concretely, the rate-limit from source VM X to destination VM Y is set to $R^{X\rightarrow Y}$:

$$\mathbf{R^{X\rightarrow Y} = max(B^{X\rightarrow Y}, R_{W\_TCP}(B^{X\rightarrow Y}, F^{X\rightarrow Y}))}$$

where R_{W_TCP} is the rate given by a weighted TCP-like algorithm operating with weight $B^{X\rightarrow Y}$ and congestion feedback $F^{X\rightarrow Y}$.

Weighted TCP-like algorithms have been extensively studied, *e.g.,* [8, 16, 22], and any of these approaches can be used for ElasticSwitch. We use a modified version of the algorithm proposed by Seawall [22], which, in turn, is inspired by TCP CUBIC.

[3]To mark a flow as unbounded, the remote-guarantee control packets contain an additional flag bit.

Seawall also uses this algorithm to control rate-limiters in hypervisors. Similar to other proposals [12, 20, 22], we use control packets to send congestion feedback from the hypervisor of the destination back to the hypervisor of the source.

The Seawall algorithm increases the rate-limit of the traffic from X to Y on positive feedback (lack of congestion) proportional to the weight, using a cubic-shaped[4] function to approach a goal rate and then explore higher rates above the goal. The goal rate is the rate where congestion was last experienced. Before reaching the goal, the rate r is increased in a concave shape by $w \cdot \delta(r_{goal} - r)(1 - \Delta t)^3$ at each iteration, where w is the weight of the flow, δ is a constant, and Δt is proportional to the time elapsed since the last congestion. Above the goal, the rate is convexly increased by $n \cdot w \cdot A$ at each iteration, where n is the iteration number and A is a constant. On negative feedback (*e.g.,* lost packets), the rate-limit is decreased multiplicatively by a constant independent of the weight.

Rationale for modifying Seawall's algorithm: Our initial approach was to actually maintain a shadow TCP-like rate and use the maximum between that rate and the guarantee. However, simply running a weighted TCP-like algorithm, such as Seawall, did not provide good results in terms of respecting guarantees. Unlike a traditional TCP-like algorithm, the rate in our case does not drop below an absolute limit given by the guarantee. When there are many VM-to-VM flows competing for bandwidth on a fully reserved link, flows would be too aggressive in poking their rate above their guarantee. For example, the rate of some VM-to-VM flows would raise above the guarantee far in the convex part of the rate increase, which would hurt the other flows.

In practice, there is a *tradeoff between accurately providing bandwidth guarantees and being work conserving*. This is particularly true since we do not rely on any switch support, and our method of detecting congestion is through packet drops. In order for flows to detect whether there is available bandwidth in the network, they must probe the bandwidth by increasing their rate. However, when the entire bandwidth is reserved through guarantees and all VMs are active, this probing affects the rest of the guarantees.

In ElasticSwitch we design the RA algorithm to prioritize the goal of providing guarantees, even under extreme conditions, instead of being more efficient at using spare bandwidth. As we show in the evaluation, ElasticSwitch can be tuned to be more aggressive and better utilize spare bandwidth, at the expense of a graceful degradation in its accuracy in providing guarantees.

Improved rate allocation algorithm: When many VM-to-VM flows compete on a fully reserved link, even a small increase in the rate of each flow can affect the guarantees of the other flows. This effect is further amplified by protocols such as TCP, which react badly to congestion, *e.g.,* by halving their rate. Thus, the algorithm must not be aggressive in increasing its rate.

Starting from this observation, we devised three improvements for the rate-adaptation algorithm, which are key to our algorithm's ability to provide guarantees:

1. *Headroom*: There is a strictly positive gap between the link capacity and the maximum offered guarantees on any link. Our current implementation uses a 10% gap.
2. *Hold-Increase*: After each congestion event for a VM-to-VM flow, we delay increasing the rate for a period inversely proportional to the guarantee of that flow.
3. *Rate-Caution*: The algorithm is less aggressive as a flow's current rate increases above its guarantee.

[4]Seawall's function is not cubic; the convex part is quadratic while the shape of the concave part depends on the sampling frequency.

Hold-Increase: After each congestion event, the hypervisor managing a VM-to-VM flow with guarantee $B^{X \to Y}$ reduces the flow's rate based on the congestion feedback, and then holds that rate for a period $T^{X \to Y}$ before attempting to increase it. This period is set inversely proportional to the guarantee, *i.e.,* $T^{X \to Y} \propto \frac{1}{B^{X \to Y}}$. Setting the delay inversely proportional to the guarantee ensures that (i) all flows in a stable state are expected to wait for the same amount of time regardless of their guarantee, and (ii) the RA algorithm still converges to rates proportional to guarantees.

Two flows, $X \to Y$ and $Z \to T$, with rates $R^{X \to Y}$ and $R^{Z \to T}$ should experience congestion events in the ratio of $C_{ratio} = \frac{R^{X \to Y}}{R^{Z \to T}}$. In a stable state, the RA algorithm ensures that rates of flows are in proportion to their guarantees: $\frac{R^{X \to Y}}{R^{Z \to T}} = \frac{B^{X \to Y}}{B^{Z \to T}}$. Thus $C_{ratio} = \frac{B^{X \to Y}}{B^{Z \to T}}$. Since the number of delay periods is proportional to the number of congestion events and the duration of each period is inversely proportional to the guarantees, both flows are expected to hold increasing rates for the same amount of time.

The delay is inversely proportional to the guarantee, rather than to the rate, to allow the RA algorithm to converge. Assuming the same two flows as above, when the $X \to Y$ flow gets more than its fair share, *i.e.,* $R^{X \to Y} > R^{Z \to T} \cdot \frac{B^{X \to Y}}{B^{Z \to T}}$, the $X \to Y$ flow is expected to experience a larger number of packet losses. For this reason, it will have more waiting periods and will be held for more time than when at its fair rate. This will allow $Z \to T$ to catch up towards its fair rate.

In cases when a large number of packets are lost, the rate of a large flow can be held static for a long time. For our prototype and evaluation, we also implemented and tested a scheme where the delay is computed in proportion to a logarithmic factor of the congestion events instead of the linear factor described above. We choose the base such that the holding periods are inversely proportional to guarantees. This approach allows RA to recover faster than the linear version after a large congestion event. However, for the specific cases we have tested in our evaluation, both the linear and the exponential decay algorithms achieved similar results, due to absence of large congestion events.

Rate-Caution: If the RA algorithm without Rate-Caution would increase the rate by amount V (there was no congestion in the network), with Rate-Caution that value is:

$$V' = V \cdot \max\left(1 - C\frac{R^{X \to Y} - B^{X \to Y}}{B^{X \to Y}}, C_{min}\right)$$

where C and C_{min} are two constants. In other words, V' decreases as the flow's current rate increases further above its guarantee. C controls the amount of cautioning, *e.g.,* if C=0.5 then the aggressiveness is halved when the rate is twice the guarantee. We use a minimum value (C_{min}) below which we do not reduce aggressiveness; this enables even VM-to-VM flows with small guarantees to fully utilize the available capacity.

Rate-Caution accelerates convergence to fairness compared to a uniform aggressiveness. When two flows are in the convergence zone, they are equally aggressive. When one flow is gaining more than its fair share of the bandwidth, it is less aggressive than the flows getting less than their fair share, so they can catch up faster. In this way, Rate-Caution allows new flows to rapidly recover the bandwidth used by the opportunistic flows using the spare capacity. The downside of Rate-Caution is lower utilization, since it takes longer to ramp up the rate and utilize the entire capacity.

Alternatives: We have experimented with multiple versions of the basic RA algorithm besides Seawall, and many algorithms achieved similar results in our tests (*e.g.,* instead of the concave-shaped sam-

pling mechanism of Seawall we used a linear function or encoded the absolute expected values for the cubic rate). In the end, we decided to use the Seawall-based one for simplicity, brevity of explanation, and since it has been successfully used by other researchers. In the future, one could also extend RA to also take into account the current round trip time (similar to TCP Vegas or TCP Nice [26]), since latency can be a good indicator of congestion.

TCP Interaction: The period of applying RA is expected to be an order of magnitude larger than datacenter round trip times (10s of ms vs. 1 ms), so we do not expect RA to interact badly with TCP.

6. IMPLEMENTATION

We implemented ElasticSwitch as a user-level process that controls `tc` rate-limiters in the Linux kernel and also controls a kernel virtual switch. We use Open vSwitch [17], which is controlled via the OpenFlow protocol. Our current implementation has ∼5700 lines of C++ code.

ElasticSwitch configures a `tc` rate-limiter for each pair of source-destination VMs. For outgoing traffic, we impose specific limits; for incoming traffic, we use the limiters only to measure rates. We configured Open vSwitch to inform ElasticSwitch of new flows and flow expirations.

ElasticSwitch uses UDP for control messages sent from destination hypervisors to source hypervisors. Given a sender VM X and a receiver VM Y and corresponding hypervisors $H(X)$ and $H(Y)$, we have two types of control messages: (1) messages from $H(Y)$ to inform $H(X)$ of the guarantee assigned for the flow from X to Y, $B_Y^{X\to Y}$, and (2) congestion feedback (packet drop counts) messages to inform $H(X)$ that there is congestion from X to Y. In our current setup, hypervisors exchange messages by intercepting packets sent to VMs on a specific control port.

In order to detect congestion, we modified the kernel virtual switch to add a sequence number for each packet sent towards a given destination. We include this sequence number in the IPv4 *Identification* header field, a 16-bit field normally used for assembly of fragmented packets. We assume that no fragmentation occurs in the network, which is typically the case for modern datacenters. A gap in the sequence numbers causes the destination to detect a congestion event and send a feedback message back to the source. To avoid a large number of congestion feedback messages (and kernel to userspace transfers) during high congestion periods, we implemented a cache in the kernel to aggregate congestion events and limit the number of messages (currently, we send at most one message per destination every 0.5ms). This patch for Open vSwitch is ∼250 lines of C code.

We have also adapted ElasticSwitch to detect congestion using ECN, as an alternative to our sequence-number scheme. If available, ECN improves congestion detection, and the ability to provide guarantees and be work-conserving (§7). We disabled ECN on the hosts, such that ElasticSwitch's congestion signaling does not interfere with TCP, and we set the ECN capable bits on all protocols (*e.g.,* UDP). For this purpose, we created another patch for Open vSwitch, which also triggers congestion feedback based on ECN bits at destinations. For accurate congestion detection, we set the ECN marking probability in switches to a high value (100%), when the queue is above a given threshold.

7. EVALUATION

The goals of this evaluation are to: (1) show that ElasticSwitch provides guarantees under worst case scenarios, and identify its limitations, (2) show that ElasticSwitch is work-conserving (*i.e.,* can improve utilization when some VMs are not active), (3) explore ElasticSwitch's sensitivity to parameters, and (4) quantify ElasticSwitch's overhead in terms of CPU, latency and bandwidth.

Summary of results: Our experiments show that:

- ElasticSwitch can achieve the intended guarantees – even in the worst conditions we tested, when traffic from 300 VMs compete with traffic from a single VM, or when multiple VMs run large MapReduce jobs at the same time. Without guarantees, the completion time of these jobs could be two orders of magnitude longer.
- ElasticSwitch provides guarantees irrespective of where congestion occurs in the network.
- ElasticSwitch is work-conserving, achieving between 75-99% of the optimal link utilization. ElasticSwitch can be tuned to be more work-conserving, at the expense of a graceful degradation in the ability to provide guarantees in challenging conditions.
- ElasticSwitch's work-conservation can increase completion times for short flows, compared to static reservations, by at most 0.7ms; however, ElasticSwitch's additional latency is no worse than when the link is fully utilized.
- ElasticSwitch is not sensitive to small changes in parameters.
- ECN support improves all results, and also makes our improvements to the rate allocation algorithm less relevant.

Experimental setup: We used ∼100 servers from a larger testbed (the actual number varied in time). Each server has four 3GHz Intel Xeon X3370 CPUs and 8GB of memory. We use a parallel, isolated network for our experiments. This prevents interference between our experiments and other traffic in the testbed. The parallel network is a two-level, single-rooted tree; all links are 1Gbps. By choosing different subsets of servers, we can create different oversubscription ratios.

Our testbed did not have ECN capable switches. However, we set up a one-rack testbed with a single ECN-capable switch.

To avoid virtualization overheads, we emulate multiple VMs by creating multiple virtual interfaces, each with its own IP address, connected to the kernel's virtual switch. Each workload generator on a host binds to a different IP address, thus emulating VMs with virtual interfaces.

We compare ElasticSwitch with two other approaches: (i) *No-Protection*: sending traffic directly, with no bandwidth protection, and (ii) *Oktopus-like Reservation*: a non-work-conserving reservation system, with rates statically set to be *optimal* for the given workload in order to achieve the hose model. Thus, Oktopus-like Reservation is an idealized version of Oktopus; in practice, Oktopus is likely to perform worse than this idealized version. We also make qualitative comparisons with Gatekeeper [20] and EyeQ [12], since those approaches cannot provide guarantees when the network core is congested.

Parameters: We use a 10% headroom between the link capacity and the maximum allocated guarantees on that link. We set the rate allocation period to 15ms and the guarantee partitioning period to 60ms. For Seawall: we use as weight $w = 450Mbps/B^{X\to Y}$, we set the rate decrease constant $\alpha = 0.4$ (so the rate is decreased to 60% after a congestion event), $\delta = 0.75$, the rate-increase constant $A = 0.5$Mbps (§5), and we scale the physical time by $T_S = 1.5$, *i.e.,* the value $\Delta t = dt/T_S$, where dt is the physical time difference. For Rate-Caution we use $C_{min} = 0.3$ and $C = 0.5$. We implemented Hold-Increase using an exponential decay of packet loss history, with decay factor γ^w, where $\gamma = 0.75$.

7.1 Guarantees and Work Conservation

We show that ElasticSwitch provides bandwidth guarantees and is work-conserving, and that ElasticSwitch provides guarantees in

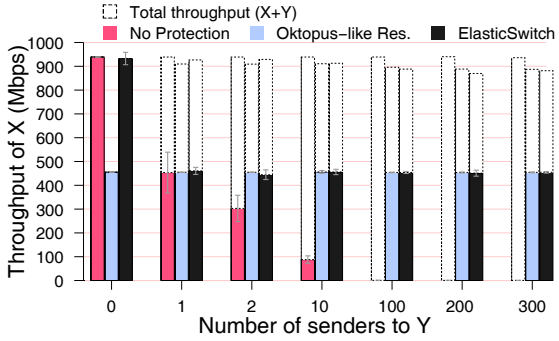


Figure 5: Many-to-one in the core. VM X receives from one remote VM while Y receives from multiple VMs. Both tenants have a guarantee of 450Mbps over the congested core link.

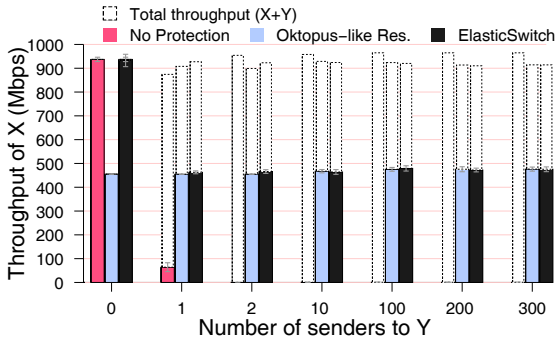


Figure 6: Many-to-one UDP vs. TCP. Same setup as Fig. 5, but senders to Y blast UDP traffic.

challenging conditions, when the entire network capacity is reserved and all VMs are fully using their guarantees. ElasticSwitch offers guarantees in all other less-congested conditions (which we do not show for brevity).

Many vs. One scenario: Two VMs X and Y that belong to two different tenants compete for a given link. Y receives traffic from multiple sources (*e.g.,* Y is a MapReduce reducer), while X receives traffic only from a single remote VM. We assume both X and Y have the same hose bandwidth guarantees of 450Mbps. Given our 10% slack in providing guarantees, these represent the maximum guarantees that can be offered on a 1Gbps link.

Fig. 5 presents the application-level TCP throughput for VM X as we vary the number of VMs sending TCP traffic to VM Y. The white bars represent the total throughput in the respective setups. For Fig. 5, X and Y are located on different servers and the congestion occurs on a core network link. Fig. 6 presents results for a different scenario, in which X and Y are on the same server and senders to Y blast UDP flows that are unresponsive to congestion. (For brevity, we omit other combinations of TCP/UDP traffic and congestion on edge/core, which exhibit similar results.) We ran the experiment for 30 seconds and X uses a single TCP flow.

Figures 5 and 6 show that ElasticSwitch provides the intended guarantees, even when the number of senders to Y is very high, and, at the same time, ElasticSwitch is able to give X the entire link capacity when no VMs are sending traffic to Y. VM Y also achieves its guarantee, as shown by the plotted total throughput; however, for more than 100 senders, TCP's efficiency in utilizing the link decreases, since some of Y's flows experience drops and timeouts and do not always fully utilize their allocated guarantees.

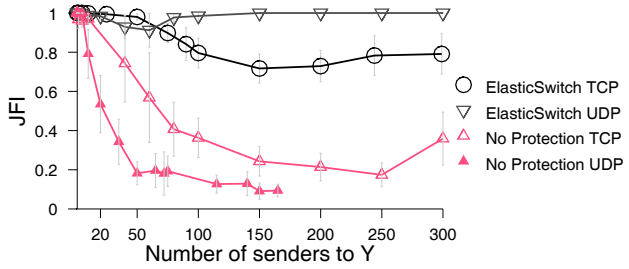


Figure 7: Many-to-one fairness. Fairness between the flows sending traffic to Y in Fig. 5 and Fig. 6.

For the scenario in Fig. 5, since the congestion does not occur on the destination server's access link, Gatekeeper [20] and EyeQ [12] would not be able to provide guarantees; we believe these solutions would perform like NoProtection in such scenarios. Seawall [22] would also perform like NoProtection for all many-to-one settings.

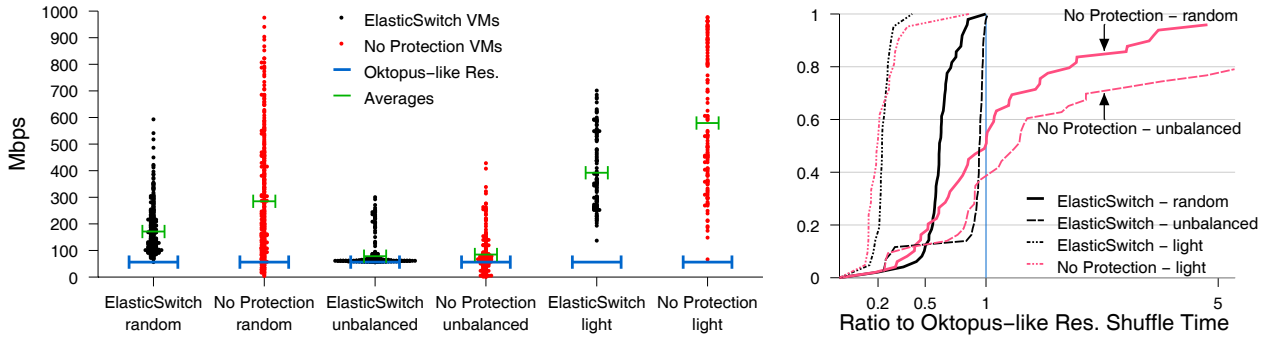
Fig. 7 shows the Jain's Fairness Index computed between the application level throughput of the flows sending traffic to VM Y. We can see that ElasticSwitch achieves better fairness than regular TCP and also provides fairness when senders use UDP flows.

MapReduce scenario: We emulate just the shuffle phase of MapReduce jobs, and measure throughputs and completion times. For easier interpretation of the results, we use a subset of the testbed, such that the topology is symmetric—*i.e.,* the oversubscription to the core is the same for all hosts. We use 44 servers where network has an oversubscription ratio of 4:1. We use 4 VM slots per server, for a total of 176 VMs.

We create multiple tenants, with random sizes from 2 to 30 VMs; half of the VMs act as mappers and half as reducers. All VMs of each tenant are provisioned with the same hose-model bandwidth guarantee, equal to the fair share of the bandwidth to the root of the topology. This translates into a guarantee of 56.25Mbps (with 10% headroom). We test with two different placement strategies: (i) "random": all VMs of all tenants are uniformly randomly mapped to server VM slots, and (ii) "unbalanced": mapper VMs of tenants are placed starting from the left corner of the tree and reduce VMs are placed starting from the right corner of the tree. The "unbalanced" case stresses the core of the network. We also test a "light" case, where fewer tenants are created, such that about 10% of the total VM slots are filled, with "random" VM placement. We use a single TCP flow between a mapper and a reducer of a tenant.

Fig. 8(a) plots the throughput recorded by each individual reducer when all jobs are active. The horizontal bars at 56.25Mbps denote the throughput achieved with a non-work-conserving system like Oktopus. As one can see, ElasticSwitch fully satisfies the guarantees in all cases (the throughput is never lower than the reservation). As expected, when using NoProtection, many VMs get less than the desired guarantees. In fact, for the "unbalanced" setup, 30% of the VMs achieve lower throughputs, some as low as 1% of the guarantee value.

Fig. 8(a) also shows that ElasticSwitch exploits unused bandwidth in the network and achieves significantly higher throughputs than an Oktopus-like static reservation system. Even in the case when all VMs are active, and not all VMs on the same machine are mappers or reducers, with MapReduce's unidirectional demand, there is unutilized bandwidth in the network (in the "unbalanced" scenarios there is very little available bandwidth). However, the average throughput achieved by ElasticSwitch is lower than NoPro-



(a) Throughput of individual reducers

(b) CDF of shuffle time under worst case assumptions

Figure 8: MapReduce experiment. (a) shows the throughput of each individual reducer. (b) shows the ratio between the worst case completion time of job (assuming the background traffic remains the same for the duration of the job) to the lower bound completion time resulted from using the guarantees.

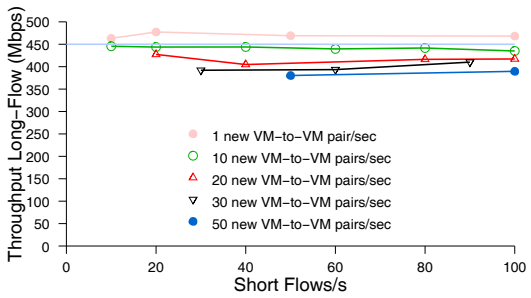


Figure 9: Short-flow impact on long flows

tection (by 25% on average), because ElasticSwitch is significantly less aggressive than TCP, so that it can always meet the guarantees.

Note that, since this is an oversubscribed topology, congestion occurs almost always in the core, and systems like Gatekeeper [20] and EyeQ [12] would be unable to always satisfy the guarantees.

We observed similar behavior for different oversubscription ratios, ranging from $1\times$ up to $5\times$ (not shown, for brevity).

Since we do not possess a full MapReduce trace, we do not try to accurately estimate the effect of ElasticSwitch on job completion time. Instead, we estimate the shuffle completion time under the worst-case assumption: the background traffic (*i.e.,* the traffic of the other jobs) remains the same for the entire duration of a job. This should be true for small jobs, but not necessarily for long jobs, for which the benefits of bandwidth guarantees might be reduced.

Fig. 8(b) plots the CDF of the ratio between the shuffle completion time using ElasticSwitch and the shuffle time obtained with Oktopus-like Reservations, under this worst-case assumption. Completion times in ElasticSwitch never exceed the completion time of static reservations (represented by the vertical line at 1), but jobs can complete significantly faster, as happens when 10% or even 100% of the VMs are active. When using no guarantees, the worst case job completion time can be up to $130\times$ longer in this simple experiment (for the "unbalanced" setup).

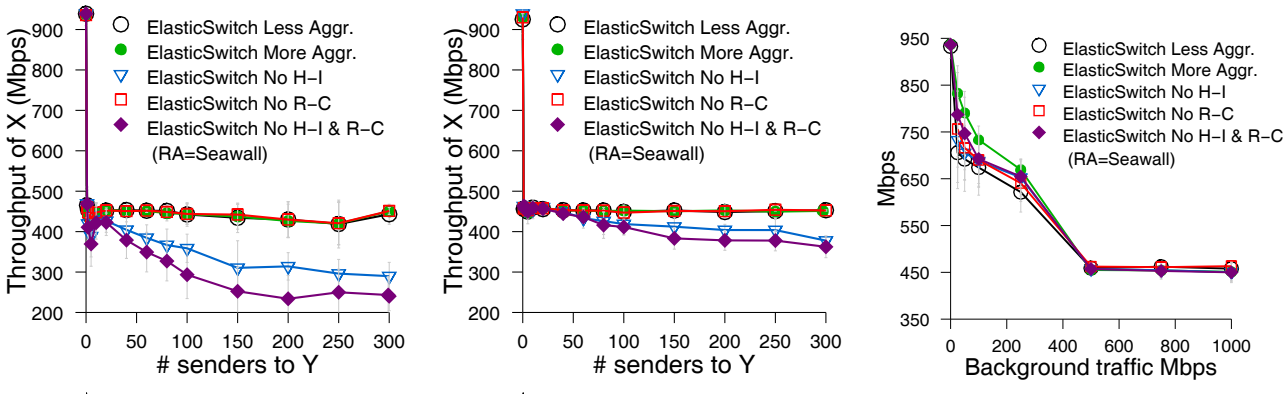
Mice vs. Elephant Flows: In this experiment, we evaluate the effectiveness of GP under large flow rates. Note that GP tries to partition guarantees across all VM-to-VM flows, to ensure that a VM can transmit/receive at its hose model guarantee. To stress GP, we use a workload with varying number of short TCP flows (mice) created at different flow rates and measure their impact on an elephant flow.

We set up two VMs X and Y with 450Mbps hose model guarantees that compete over one 1Gbps link L (thus L is fully reserved). Each of X and Y send traffic using a single elephant TCP flow to remote VMs (*e.g.,* X sends to VM Z while Y sends to VM T). In addition to this flow, VM X generates a number of short TCP flows (HTTP requests) to other VMs.

We start from the observation that the guarantee can be "wasted" on short flows only if the short flows are always directed to a different VM. However, actual data (*e.g.,* [7]) shows that the number of services to which one service communicates are very small, and thus the number of actual VMs with which one VM exchanges short flows in a short time should be small.

Fig. 9 shows the throughput achieved by the elephant flow of VM X for various cases. We vary the total number of short flows X generates per second from 10 to 100, out of which 10 to 50 are new VM-to-VM flows. Results show that below 10 new VMs contacted by X every second, there is little impact on X's elephant flow, regardless of the number of mice flows generated by X. For 20 or more new VMs contacted by X, there is an impact on the throughput of X's long flow. However, we note that even if a VM were to contact so many new VMs per second with short flows, it cannot sustain this rate for too long, since it would end up communicating with thousands of VMs in a short time. For this reason we believe the impact of short flows on the throughput of elephant flows in ElasticSwitch is not a significant concern.

Sharing Additional Bandwidth: We show that ElasticSwitch shares the additional bandwidth roughly in proportion to the guarantees of the VM-to-VM flows competing for that bandwidth. In this experiment, two flows share a 1Gbps link, with one flow given a guarantee that is twice the guarantee of the other. We measure the resulting throughputs to examine how the residual bandwidth on the link (left over after the guarantees are met) is shared. Table 2 presents these results for three cases. As expected, since the cumulative guarantee is less than 900Mbps, both flows achieve higher throughput than their guarantees. Ideally, the flow with higher guarantee should achieve 2X higher throughput than the other flow and they should fully utilize the 1Gbps link. However, the flow with the higher guarantee grabs a slightly disproportional share of the residual bandwidth, with the non-proportionality increasing with larger residual link bandwidth. We further analyzed our logs and noticed that the ratio of the dropped packets did not exactly follow the ratio of the rates, but was closer to a 1:1 ratio. This causes rate allocation to hold the rate of the flow with smaller guarantee for longer periods. This is a place to improve the RA algorithm in the future.



(a) Many-to-one 1 TCP flow (b) Many-to-one 2 TCP flows (c) Throughput vs. UDP Background Traffic

Figure 10: Sensitivity vs. various parameters.

Guar. (Mbps)	45	90	90	180	225	450
Rate (Mbps)	203	536	219	531	275	589
Ratio	2.64		2.42		2.14	

Table 2: Sharing a 1Gbps link between two VM-to-VM flows with one guarantee being twice the other

7.2 Sensitivity

Aggressiveness: In general, by tuning the RA algorithm to be more aggressive in grabbing spare bandwidth, ElasticSwitch can better utilize the available capacity. On the other hand, a less aggressive RA makes ElasticSwitch better suited to provide guarantees in difficult conditions. We now show that ElasticSwitch is resilient to parameter changes, and that Hold-Increase and Rate-Caution proved useful for providing guarantees in difficult conditions.

Fig. 10 shows the behavior of ElasticSwitch for different rate allocation algorithms. By "Less Aggr." we refer to ElasticSwitch using the parameters described at the start of this section. By "More Aggr." we refer to ElasticSwitch using more aggressive rate increase parameters for Seawall (specifically we use a rate increase constant of 2.5Mbps for the convex curve instead of 0.5Mbps, and the time scalar T_S=2.0 instead of 1.5 for the "Less Aggr." case). By "No H-I" we refer to not using Hold-Increase, by "No R-C" we refer to not using Rate-Caution, and by "No H-I & R-C" we refer to not using either of them. Note that not using Hold-Increase nor Rate-Caution is equivalent to applying Seawall's algorithm for rate allocation. All of the last three RA algorithms use the less aggressive Seawall parameters; however, by not using Hold-Increase and/or Rate-Caution they become (significantly) more aggressive.

Figures 10(a) and 10(b) show a scenario similar to that in Fig. 5 (with the minor difference that the congestion occurs on the edge link in this experiment, which, surprisingly, proved more challenging). However, while in Fig. 10(a) all VMs use a single TCP flow, in Fig. 10(b), all VM to VM traffic consists of two TCP flows. Two TCP flows better utilize the bandwidth allocated by the RA algorithm. We observe that increasing the aggressiveness of ElasticSwitch is almost unnoticeable, showing that ElasticSwitch is quite resilient to parameter changes. We can also see that Hold-Increase is instrumental for achieving guarantees when congestion is detected through packet losses. Rate-Caution proves helpful for more aggressive approaches such as Seawall, though its effect on the non-aggressive ElasticSwitch is small.

Fig. 10(c) shows the other face of being less aggressive, using the same TCP vs. UDP scenario as in Fig. 2. When the UDP flow's demand is below its guarantee of 450Mbps, the TCP flow attempts to grab the additional bandwidth. As one can see, in this case being more aggressive pays off in achieving a higher link utilization.

Periods: Changing the RA and GP periods did not significantly affect the results in our experiments. We experimented with rate allocation periods from 10ms to 30ms and with guarantee partitioning periods from 50 to 150ms. For brevity, we omit these charts. It is true, however, that the workloads we considered in this evaluation do not stress throughputs that vary significantly in time, except for short flows. Thus, exploring the effects of different periods is also a place for future work.

7.3 Overhead

We evaluate the overheads of ElasticSwitch in terms of latency, CPU, and control traffic bandwidth.

Latency: Compared to NoProtection, ElasticSwitch adds latency due to the rate limiters in the data path and a user-space control process that sets up the limiters for each new VM-to-VM flow. Compared to the Oktopus-like Reservation, ElasticSwitch adds overhead due to the filling of queues in the intermediate switches for work conservation. Fig. 11 shows the completion time of a short flow generated by VM X towards VM Y in several circumstances: (a) when there is no traffic (and this is the first flow between X and Y, *i.e.,* a "cold" start); and (b) when we vary the number of VM-to-VM flows that congest a given link L that the short flow traverses. We vary this number from one up to the maximum number of other VMs that share the guarantees on link L with X and Y (23 other VM-to-VM flows in this case; we use a 6× oversubscribed network with 4 VMs per server). We use a single TCP flow between each VM pair.

Fig. 11 shows that ElasticSwitch increases completion time only slightly when there is no other traffic. With background traffic, the added latency of ElasticSwitch is smaller than when using NoProtection, because rate allocation algorithm is less aggressive than the TCP flows, and queues have lower occupancy. However, this delay is always larger than a static reservation system, which keeps links under-utilized.

CPU: Since ElasticSwitch adds overhead in both kernel and userspace, and since tested applications also consume CPU resources, we only evaluate the difference in CPU usage between ElasticSwitch and NoProtection in the same setup. (It is hard for us to estimate the overhead of an Oktopus-like Reservation solution since it could be implemented in several ways.)

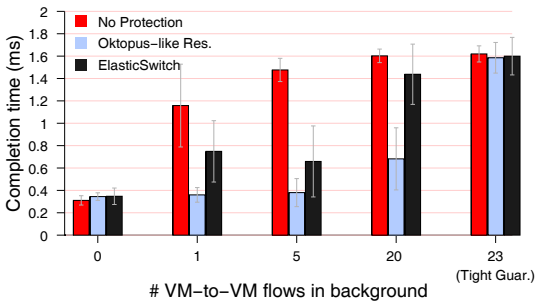


Figure 11: Completion time of HTTP requests vs. background traffic

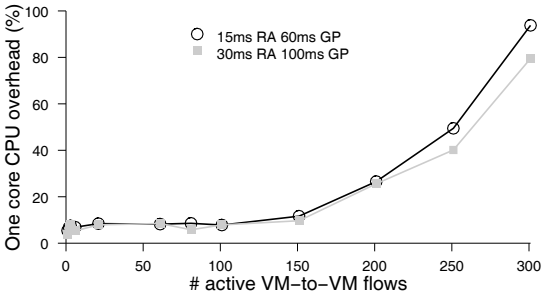


Figure 12: CPU Overhead vs. # Active Flows

Fig. 12 shows the overhead of ElasticSwitch compared to No-Protection, in terms of the capacity of a single CPU core (our implementation is multi-threaded), as we vary the number of active VM-to-VM flows. These results were measured in the context of the many-to-one experiment where the two VMs X and Y are located on the same server (*e.g.,* the results presented in Fig. 10(a)). Because a part of the overhead depends on the frequency of applying GP and RA, we plot two different sets of periods. Our fine-grain profiling indicates that most of the CPU cycles are spent in reading and setting the rate-limiters; after a certain number of limiters, this overhead seems to start to increase nonlinearly.

Fig. 12 shows that the additional CPU overhead of ElasticSwitch can be handled in typical cases with one CPU core (we note also that our testbed uses older generation CPUs). We believe this overhead can be significantly improved in the future, for example, by using an improved rate-limiting library.

Control Traffic: ElasticSwitch uses two types of control packets: remote guarantees and congestion feedback, both having the minimum Ethernet packet size (64B). For a GP period of 60ms (used in our prototype), ElasticSwitch sends roughly 17 control packets per second for each VM-to-VM active flow; *e.g.,* if there are 100 communicating VM-to-VM pairs on one server, the traffic overhead for sending/receiving remote guarantees is ∼850Kbps. In the current implementation, ElasticSwitch sends one congestion feedback control packet for each congestion event, limited to at most one message per 0.5ms. However, since ElasticSwitch detects only the packets effectively lost inside the network (and not in the queue of the sending host), and since RA is not aggressive and keeps buffers free, this traffic is very small—on the order of few Kbps.

7.4 ECN

Our limited experience on the single ECN-capable switch setup suggests that: (i) ECN improves the results both for enforcing guarantees and for being work conserving, and (ii) our improvements to Seawall's algorithm are not necessary on ECN-capable networks. In brief, the many-to-one experiment results were ideal in terms of providing guarantees, showing very little variance (we could test up to 100 senders). For the experiment of borrowing bandwidth from a bounded flow (Fig. 2 and Fig. 10(c)), ECN would improve results, being similar to the more aggressive ElasticSwitch in Fig. 10(c). For space constraints, we do not plot these results.

8. RELATED WORK

Oktopus [4] provides predictable bandwidth guarantees for tenants in cloud datacenters. Oktopus does both placement of VMs and enforcement of guarantees. However, the enforcement algorithm is not work-conserving and hence Oktopus does not utilize the network efficiently. Further, the approach is centralized and thus has limited scalability.

SecondNet [11] provides a VM-to-VM bandwidth guarantee model in addition to the hose model. In SecondNet, a central controller determines the rate and the path for each VM-to-VM flow and communicates those to the end hosts. As with Oktopus, SecondNet's guarantee enforcement is not work-conserving and has limited scalability, because of its centralized controller. Further, it requires switches with MPLS support.

Gatekeeper [20] and EyeQ [12] also use the hose model, are fully implemented in hypervisors and are work conserving. They are also simpler than ElasticSwitch. However, these approaches assume that the core of the network is congestion-free: they can provide guarantees only in that case. However, research shows that in current datacenters congestion occurs almost entirely in the core [6, 24].[5] EyeQ also requires switches with ECN support, to detect congestion at the Top-of-the-Rack switches.

FairCloud [18] analyzes the tradeoffs in allocating cloud networks and proposes a set of desirable properties and allocation policies. One of FairCloud's bandwidth-sharing proposals, PS-P, provides bandwidth guarantees and is work-conserving. However, FairCloud's solution requires expensive hardware support in switches (essentially one queue for each VM) and works only for tree topologies.

Seawall [22] uses a hypervisor-based mechanism that ensures per-source fair sharing of congested links. NetShare [13] ensures per-tenant fair sharing of congested links. However, neither approach provides any bandwidth guarantees (*i.e.,* the share of one VM can be arbitrarily reduced [18]). For this reason, these sharing models cannot be used to predict upper bounds on the runtime of a cloud application.

Proteus [27] proposes a variant of the hose model, Temporally-Interleaved Virtual Cluster (TIVC), where the bandwidth requirements on each virtual link are specified as a time-varying function, instead of the constant in the hose model. Proteus profiles MapReduce applications to derive TIVC models. Proteus uses these models for placement and enforcement of bandwidth guarantees. Thus Proteus is suited for a limited set of applications; it is not workload-agnostic, as is necessary for cloud computing.

Hadrian [5] is a recent proposal focusing on providing bandwidth guarantees for inter-tenant communication. ElasticSwitch can use Hadrian's pricing and reservation schemes for inter-tenant communication. The mechanism used by Hadrian to enforce bandwidth guarantees has some resemblance to ElasticSwitch at a high level. However, Hadrian's approach requires dedicated switch support for setting flow rates in data packets, unavailable in today's hardware.

[5]The core is not guaranteed to be congestion-free even for fully-provisioned networks. Since these networks are not cliques, extreme many-to-one traffic patterns can congest links inside the core.

9. DISCUSSION AND FUTURE WORK

ElasticSwitch on Other Topologies: ElasticSwitch can be used on any single-path routing topology, as long as the admission control criterion is respected. ElasticSwitch can also be applied to multi-path topologies, where load balancing is uniform across paths, such as fat-trees [1] or VL2 [10]. In this case, the set of links on which traffic is load balanced can be seen as a single generalized link, *e.g.,* for fat-trees, a generalized link represents the set of parallel links at a given level (*i.e.,* that can be used by one VM towards the root nodes). We are working towards deploying a load balancing solution and testing ElasticSwitch on top of it.

For multi-path topologies with non-uniform load balancing across paths, such as Jellyfish [23], ElasticSwitch could be extended to use three control layers instead of two: guarantee partitioning, *path partitioning* and rate allocation. Path partitioning would divide the guarantee between two VMs among multiple paths. The other layers would operate unmodified.

Prioritizing Control Messages: Since it is applied periodically, ElasticSwitch is resilient to control message losses (we have tested this case). However, the efficiency of ElasticSwitch is reduced, particularly at high loss rates. Ideally, control messages should not be affected by the data packets, and one could ensure this by prioritizing control packets, *e.g.,* using dedicated switch hardware queues. (In our current implementation, ARP packets supporting the control traffic should also be prioritized or ARP entries pre-populated; however, we believe this can be avoided in future versions.)

Limitations: Endpoint-only solutions for providing bandwidth guarantees are limited by the fact that they use shared queues instead of dedicated queues. For instance, a malicious tenant can create bursts of traffic by synchronizing packets from multiple VMs. Synchronization creates a temporary higher loss rate for the other tenants, which can negatively affect their TCP flows. We leave addressing these concerns to future work.

Beyond the Hose Model: The hose model has the advantage of simplicity. It is ideal for MapReduce-like applications with all-to-all communication patterns, but it might be inefficient for applications with localized communication between different components [4, 14]. ElasticSwitch can be used as a building block for providing more complex abstractions based on hose models, such as the TAG model [14], or a model resembling the VOC model [4]. The high level idea is that each VM can be part of multiple hose models; *e.g.,* we can have a hose model between VMs from two application components, and an additional hose-model between VMs within a single component. In this case, GP must identify the hose(s) to which a VM-to-VM flow belongs. ElasticSwitch does not support hierarchical models that aggregate multiple VM hoses into one [4, 5], as this would require coordination across VMs.

10. CONCLUSION

We have presented ElasticSwitch, a practical approach for implementing work-conserving minimum bandwidth guarantees in cloud computing infrastructures. ElasticSwitch can be fully implemented in hypervisors, which operate independently without the use of a centralized controller. It works with commodity switches and topologies with different over-subscriptions. ElasticSwitch provides minimum bandwidth guarantees with hose model abstractions—each hose bandwidth guarantee is transformed into pairwise VM-to-VM rate-limits, and work conservation is achieved by dynamically increasing the rate-limits when the network is not congested. Through our implementation and testbed evaluation, we show that ElasticSwitch achieves its goals under worst case traffic scenarios, without incurring a high overhead.

Acknowledgments: We thank the anonymous reviewers and our shepherd, Jitu Padhye, for their guidance on the paper.

11. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*. ACM, 2008.
[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.
[3] H. Ballani, P. Costa, T. Karagiannis, et al. The price is right: Towards location-independent costs in datacenters. In *Hotnets*, 2011.
[4] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *ACM SIGCOMM*, 2011.
[5] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, et al. Chatty Tenants and the Cloud Network Sharing Problem. *NSDI'13*.
[6] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*. ACM, 2010.
[7] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving Failures in Bandwidth-Constrained Datacenters. In *SIGCOMM*, 2012.
[8] J. Crowcroft and P. Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *SIGCOMM CCR*, July 1998.
[9] N. G. Duffield, P. Goyal, A. G. Greenberg, P. Mishra, K. Ramakrishnan, and J. E. van der Merwe. A Flexible Model for Resource Management in VPNs. In *SIGCOMM*, 1999.
[10] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. *ACM SIGCOMM*, 2009.
[11] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *CoNEXT*. ACM, 2010.
[12] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *USENIX NSDI*, 2013.
[13] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese. NetShare: Virtualizing Data Center Networks across Services. *UCSD TR*, 2010.
[14] J. Lee, M. Lee, L. Popa, Y. Turner, P. Sharma, and B. Stephenson. CloudMirror: Application-Aware Bandwidth Reservations in the Cloud. *HotCloud*, 2013.
[15] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center Ethernet for multipathing over arbitrary topologies. In *USENIX NSDI*, 2010.
[16] T. Nandagopal, K.-W. Lee, J.-R. Li, and V. Bharghavan. Scalable Service Differentiation using Purely End-to-End Mechanisms: Features and Limitations. *Computer Networks*, 44(6), 2004.
[17] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, et al. Extending Networking into the Virtualization Layer. In *HotNets'09*.
[18] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *ACM SIGCOMM*, 2012.
[19] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM*, 2011.
[20] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *USENIX WIOV*, 2011.
[21] R. Sherwood, G. Gibb, K.-K. Yap, M. Casado, N. Mckeown, et al. Can the production network be the testbed? In *OSDI*, 2010.
[22] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *Usenix NSDI*, 2011.
[23] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *USENIX NSDI*, 2012.
[24] Srikanth K and Sudipta Sengupta and Albert Greenberg and Parveen Patel and Ronnie Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *IMC*. ACM, 2009.
[25] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. In *Proc. HotNets*, NY, NY, Oct. 2009.
[26] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *OSDI*, 2002.
[27] D. Xie, N. Ding, Y. C. Hu, and R. Kompela. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *SIGCOMM*, 2012.