TALON: Tenant Throughput Allocation Through Traffic Load-balancing in Virtualized Software-defined Networks

Heesang Jin, Gyeongsik Yang, Bong-yeol Yu, Chuck Yoo
Department of Computer Science and Engineering
Korea University
Seoul, Republic of Korea
{hsjin, ksyang, byyu, chuckyoo}@os.korea.ac.kr

***Abstract*—Virtualized software-defined networking (SDN) has been drawing increasing attention in data centers. SDN enables the creation of arbitrary and flexible virtual networks. However, given that the physical network is shared among multiple tenants, the throughput of a tenant can interfere with that of the other tenants, and the throughput requirement on each tenant cannot be met. To solve this problem, we propose TALON, a throughput allocation scheme based on traffic load-balancing. TALON allocates a throughput per tenant flow by calculating multiple paths to fulfill the traffic requirement. We design and implement TALON using an open-source network hypervisor. The evaluation results show that each tenant's throughput requirements are nearly met. In addition, the throughput increases by up to 2.29 times compared to that of a non-load-balancing network.**

***Index Terms*—Network virtualization, Software-defined networking, Traffic load-balancing, Throughput allocation**

I. Introduction

Fully virtualized software-defined networking (vSDN) has attracted much attention from researchers because of its flexibility that enhances network management services [1]. In vSDN, a tenant can arbitrarily create its virtual network topology by designating virtual switches and ports which are then mapped to physical switches and ports. The isolation of multiple virtual networks is the task of the network hypervisor (NH), which is a key component of vSDN that resides between the physical network and tenant controllers. NH manages the mapping between the physical and virtual networks and translates each control message to ensure network isolation.

As each tenant network can create any virtual network topology on the physical network and can configure its own policy, vSDN is useful for adopting multiple services with diverse traffic characteristics within a single network, achieving high resource consolidation. So far, various NHs [2]–[4] that manage the network according to their own goals (e.g., topology virtualization, address virtualization, policy composition) have been proposed; however, only a few studies have considered the traffic performance of virtual networks.

We conduct our own preliminary experiments, in which two tenants share one forwarding path consisting of three switches, and require 200 Mbps throughput for their flow. We found that the tenants' throughputs interfere with each other; hence, the throughput requirement could not be met. This indicates that the tenant performance is not isolated. This severely reduces the quality of services provided by data centers [5], [6]. This stems from the lack of throughput management in NHs. Without any throughput allocation, the throughput cannot meet the requirements owing to the traffic from other tenants.

To solve this problem, we propose TALON, a throughput allocation scheme that applies traffic load-balancing in a vSDN. When the routing results of the tenant controller arrive at TALON, TALON determines whether the path is sufficient for the throughput requirement of the tenant. If the path is insufficient, TALON calculates additional paths to fulfill the throughput requirement. To implement this scheme, TALON consists of two main functionalities: 1) throughput allocation, which calculates packet forwarding paths for the given throughput requirement and 2) multi-path splitting, which installs flow rules to realize the calculated multi-paths in the physical network. We implement TALON on an open-source NH using the standard OpenFlow protocol.

The remainder of this paper is organized as follows. Section II elaborates on the background, motivation of this study. The Section II also presents relevant studies to TALON. The design of TALON is presented in Section III. Section IV presents the implementation and evaluation results. Finally, we present our conclusions in Section V.

II. Background and Motivation

A. Fully virtualized SDNs

Figure 1 depicts the general architecture of vSDN. All physical switches are connected to NH, and NH configures them using the control messages created from the virtual network controllers. NH functions as a controller to the physical switches and as virtual switches to the virtual network controller. When a physical switch receives a packet but does not contain any rules for processing it, the switch requests a flow rule for processing the packet to its controller. The flow rule request is first transmitted to NH, and NH delivers the packet to the corresponding virtual network (tenant) controller after translating the physical addresses into virtual addresses. Upon receiving the flow rule request, the tenant controller

calculates the forwarding path within the virtual network topology and creates a flow rule using the virtual switches belonging to the calculated path. Then, NH translates the flow rule from the tenant controller into a physical flow rule and installs the rule on the physical switch that is mapped to the virtual switch.

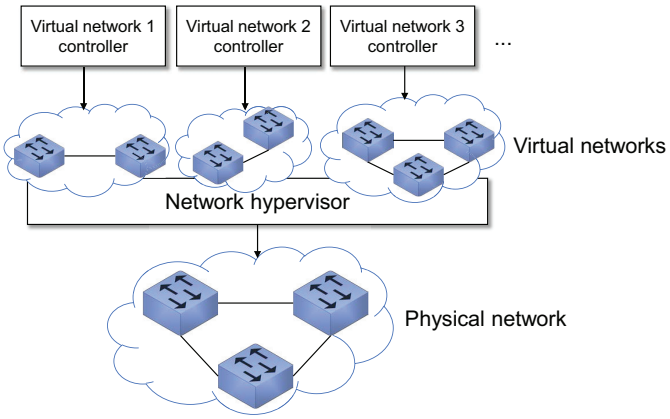


Fig. 1: SDN-based network virtualization architecture

B. Motivation and Goal

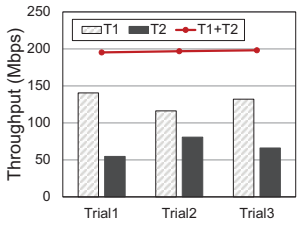


Fig. 2: Average throughput of two tenants over three trials

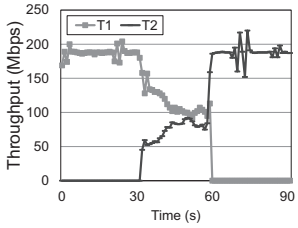


Fig. 3: Throughput of two tenants sharing the same path

We conducted a simple experiment to demonstrate the motivation for this study. Using the physical network described in Section IV, we create two virtual networks consisting of path 0 (S1–S2–S7) with [4], which is an open-source NH. Tenant 1 (T1) creates traffic first, and tenant 2 (T2) generates traffic after 30 s. Each tenant creates ten 20 MB/s TCP connections for 60 s; thus, a total of 200 MB/s is required for each. We repeat this experiment three times, and Fig. 2 shows the average throughput from 30 s to 60 s, when the traffic of both tenants overlaps. In addition, we show the change of throughput in Fig. 3. In results, the throughput for one tenant suffers because of interference from the other tenant as they share the same physical path. Thus, the desirable throughput has not been achieved.

We believe that NH should satisfy the desired throughput requirement. However, OpenVirteX [4], which is used in our experiment, and existing NHs do not provide sufficient throughput management. In vSDN, NH receives a forwarding path from the tenant controller. Then, without any throughput consideration or allocation for the given path, NH merely installs the path on the physical switch after topology and address virtualizations. Therefore, the interference of throughput between tenants is inevitable. To solve this problem, we need a scheme that allocates the throughput of flow by considering the tenant-created path. We leverage traffic load-balancing scheme for throughput allocation to install the tenant-created path and provide throughput with additional multiple paths.

C. Related studies

This paper addresses vSDN, performance management on vSDNs, and traffic load-balancing techniques. In the following subsections, we discuss these topics in detail.

1) SDN-based network virtualization: Various NHs have been studied. FlowVisor [2] enables switch slicing, which allocates a subset of the available flowspace to each virtual network. FlowN [3] enhances this by introducing address virtualization with VLAN tunneling, and OpenVirteX employs another method of address virtualization that allocates a physical address for each virtual address and rewrites the addresses at the edge switches. In addition, CoVisor [7] combines the flow rules from multiple controllers to manage a single network with heterogeneous controllers. Finally, OnVisor [8] introduces a distributed NH based on OpenVirteX.

2) Performance management on vSDN: The implementation of [4] includes the calculation and installation of backup paths for big virtual links, which are mapped to a subset of the physical network. When a new big link is created, OpenVirteX calculates and maps the shortest path between two designated physical ports, and also calculates the backup paths within the physical ports. Then, in case the mapped shortest path fails, OpenVirteX installs the flow rules for the backup path. In [9], the authors improved the implementation of OpenVirteX to reduce the number of flow rules that are updated when a failure occurs. However, no studies have attempted to provide multiple paths for a single path specified by a virtual network controller. BAS [10] proposed a periodic physical resource remapping for virtual links to enhance throughput. However, the allocation of the required amount of throughput was not discussed.

3) Throughput allocation: Various studies that allocate throughput depending on the requirements of tenants have been proposed. For example, in [11], [12], Google presented a bandwidth allocation scheme for their data center infrastructure and wide-area network that connects data centers. In addition, [13] proposed a work-conserving-based bandwidth management scheme relying on sender rate control. However, to the best of our knowledge, no study has discussed the bandwidth allocation in a vSDN. In addition, most studies provide bandwidth allocation using a routing approach. In contrast, we use a load-balancing technique.

III. Design

Here, we describe the key design components of TALON. We first show the overall architecture of TALON and then discuss each component in detail.

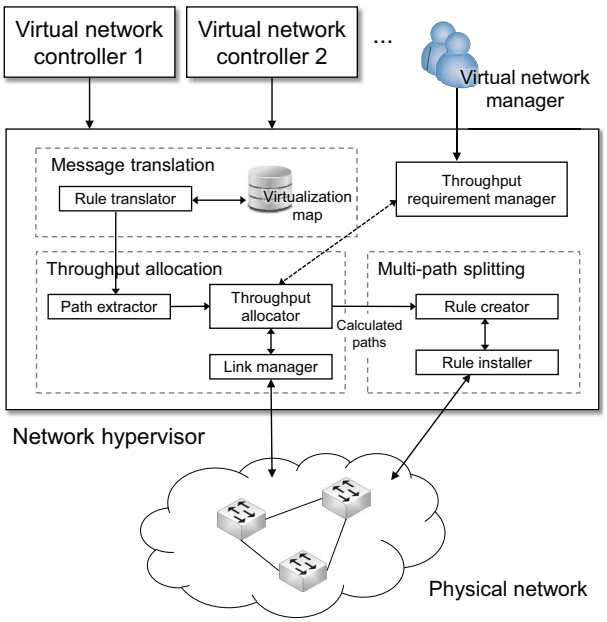


Fig. 4: TALON architecture

A. Design overview

Figure 4 shows the architecture of TALON. TALON has two main functionalities: 1) throughput allocation and 2) multi-path splitting. When a flow rule message created by a tenant controller arrives at TALON, the flow rule is first translated into a physical context, which means the target switch and virtual addresses are translated into a physical switch and physical addresses (*message translation* in Fig. 4). Then, TALON extracts the path information from the various flow rules to allocate the throughput requirement for each tenant. Subsequently, TALON allocates the throughput using multi-path calculations that consider the required throughput and available bandwidth on each link. Finally, flow rules based on the calculated paths are created, and TALON sends the flow rule messages to the corresponding physical switches.

B. Throughput allocation

To allocate the throughput required by each tenant, we apply four components. First, *link manager*, which maintains the available bandwidth on each link, and *throughput requirement manager*, which provides an API for each tenant to enter the throughput requirements and maintains the values, are applied. Then, when the tenant sends flow rules for the forwarding path, TALON creates the end-to-end path information from flow rules in *path extractor*. The information is delivered to *throughput allocator* which checks the throughput availability of the path and calculates the additional path required to fulfill the throughput requirement. We discuss each component and algorithm in detail.

1) Link manager and Throughput requirement manager: To allocate the throughput for each tenant, TALON obtains the available bandwidth from links in the physical network. *Link manager* is the component that determines the maximum bandwidth on the links in the physical network. In a typical SDN network (e.g., an OpenFlow-based network), the maximum link bandwidth can be obtained from the port configuration. TALON receives the maximum bandwidth on each link when the physical switches are set up.

Moreover, *throughput requirement manager* takes the tenant's throughput requirement. We define a simple triple syntax - {source host, destination host, throughput} for each tenant to enter their requirements on TALON. When the values are applied to TALON, *throughput requirement manager* stores the requirements in the form of a hash table.

2) Path extractor: When the forwarding path is given by the tenants, *path extractor* creates end-to-end path information from the flow rules. *Path extractor* first checks whether the same path has been created and installed previously because flow rule requests can be continuously generated during a path installation. If the path has not been created, *path extractor* creates the path information (i.e., source host, ingress switch, core switch, egress switch, and destination host) using multiple messages. Then, the path information is delivered to *throughput requirement manager*, which will be discussed in the next section.

3) Throughput allocator: Throughput allocator is the component that performs the core functions of TALON. Given a tenant-created path (main path), *throughput allocator* first calculates the remaining throughput capacity (C_R) of the path. Then, it allocates the required throughput in the following manner. First, *throughput allocator* checks if the main path alone can fulfill the throughput requirement. When C_R is large enough for the requirement, TALON omits further traffic load-balancing and installs only the main paths. On the other hand, when C_R is insufficient, TALON checks whether the intended main path is being shared by the other tenants. If the capacity of the main path is allocated to other sub-paths belonging to other tenants, *throughput allocator* deletes and reallocates the sub-paths until the link's spare capacity becomes sufficient to ensure the required throughput. Then, the secured capacity is allocated for the new main path. This is because the main path is prioritized over the sub-path. The entire sequence is summarized in Algorithm 1.

However, if the main path is only used for the other tenants' main paths or the ensured capacity from the sub-paths is still insufficient to meet the throughput requirement, TALON calculates additional paths to split the flow. We leverage the shortest-path algorithm to calculate sub-paths using C_R as the metric.

C. Multipath splitting

Based on the calculated paths from the throughput allocation, TALON installs the flow rules that can be adapted for load-balancing to enable multi-path splitting. We categorize the flow rules into three categories as follows: 1) ingress flow rules, 2) core flow rules, and 3) egress flow rules. An ingress flow rule is the first rule for traffic splitting; hence, this rule should be modified to forward the packets that arrive

Algorithm 1 Throughput Allocation

```
1: procedure INSTALLPATH
2:     MP ← Main path consisting of a set of physical
           switches
3:     R_f ← Throughput requirement for the flow
4:     RP ← Set obtained from physical paths
           to be installed at the physical network
5:     C_R^j ← Remaining capacity of the physical path j
6:     C_A^k ← Allocated capacity of the path k
7:     if C_R^MP > R_f then
8:         RP += MP
9:         Return
10:    end if
11:    if 0 < C_R^MP < R_f then
12:        R_f -= C_R^MP
13:        victimPaths = findVictimSubPath(MP)
14:        removedPaths += unInstall(victimPaths)
15:        R_f -= C_A^removedPath
16:        if R_f > 0, then
17:            subPath = shortestPath(MP)
18:            RP += subPath
19:        end if
20:    end if
21:    install(RP)
22:    if removedPaths ≠ ∅ then
23:        ReAllocate(removedPaths)
24:    end if
25: end procedure
```

Algorithm 2 Rule Creator

```
1: procedure CREATEFLOWRULES
2:     MP ← Set of switches consisting of the main path
3:     SP ← Set of switches consisting of the sub-paths
4:     C_A^k ← Allocated throughput capacity of the path k
5:     FRule ← Flow rule
6:     GRule={<C_A^k,k>} ← Group table rule splitting
           traffic for path k
7:     GRule={⟨C_A^MP, MP⟩,⟨C_A^SP, SP⟩}
8:     FRule.match =⟨srcHost, dstHost⟩
9:     FRule.action = Goto GRule
10:    Install (IngressSW, GRule)
11:    Install (IngressSW, FRule)
12:    for path of {MP, SP} do
13:        for switch in path do
14:            FRule.match = Path.label
15:            if switch == coreswitch then
16:                FRule.action = Goto next switch
17:            else
18:                FRule.action = Goto dstHost
19:            end if
20:            Install (switch, FRule)
21:        end for
22:    end for
23: end procedure
```

at the same in-port to multiple out-ports. The core flow rules refer to the intermediate rules, which are not included in the ingress or egress switches. In this case, because the packets are simply forwarded through a single in-port and out-port, such rules are not modified. An egress flow rule is the ending rule of converging traffic; hence, this rule should be modified to forward the packets that come through multiple in-ports to a single out-port.

Rule creator is a component used for flow rule creation. We deploy a group table scheme, which is a standard feature of OpenFlow, for traffic splitting and failover. This feature is supported from version 1.1 onwards. To achieve the traffic splitting with the group table, *rule creator* creates a set of flow rules and group rules based on the results of *throughput allocator*.

Algorithm 2 summarizes the overall operation of *rule creator*. For the ingress rule, *rule creator* installs a group rule that splits a single flow to be sent through multiple output ports. *Rule creator* also installs a flow rule that directs the group rule to process the packets. The flow rules for the core switches are first installed for the main path, and then for the sub-paths. Finally, TALON installs a separate flow rule for the egress switch that transmits the packets from multiple paths to the destination host.

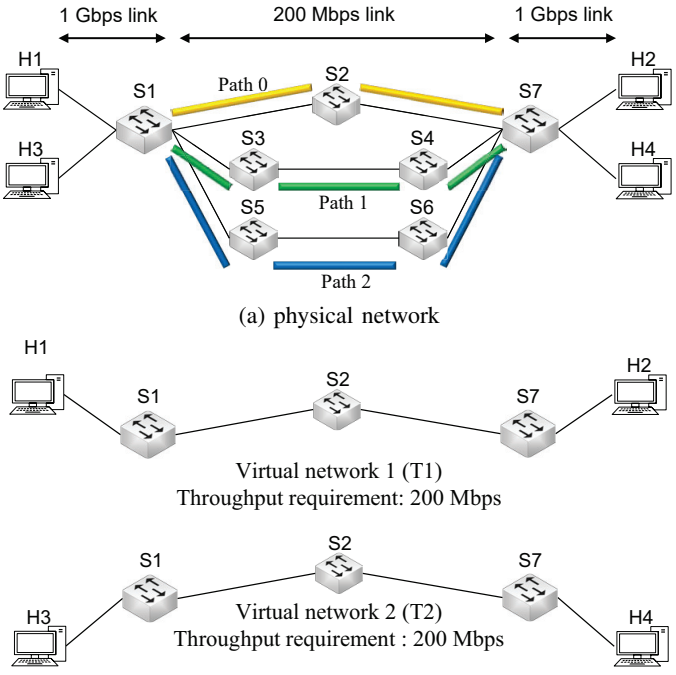
After the flow rules for traffic splitting have been created, *rule installer* sends the flow rule and group rule installation messages.

IV. Implementation and Evaluation

We implement TALON on OpenVirteX, which is an open-source NH. TALON uses a group table mechanism to implement traffic splitting. Because this mechanism is supported in OpenFlow from version 1.1 onwards, we extend the southbound interface of OpenVirteX for OpenFlow 1.0 and 1.3 interfaces. Moreover, we use the ECMP [14] load-balancing mechanism on the physical switch, which splits traffic using a hash operation. As our scheme deals with the throughput allocation at NH side, other load balancing techniques on the data plane can also be used.

We emulate the physical network using Mininet [15], as shown in Fig. 5a, and create two virtual networks that consist of switches S1, S2, and S7. One host is connected to each edge switch on the virtual networks. We set the traffic requirement for each tenant (traffic from one host to another) to 200 Mbps. Each virtual network is controlled via an ONOS controller. The Mininet, NH with TALON, and ONOS controllers are executed on separate physical machines. We use iperf3 to generate ten TCP connections from a client to server, generating 20 Mbps of traffic per connection. Thus, each tenant has a total of 200 Mbps traffic for the evaluation (Fig. 5b).

First, we generate the traffic of two tenants at the same time and evaluate the improvement in the throughput when TALON is employed. Figure 6 shows the average and standard error of the throughput per tenant over five trials. Figure 7 shows



(a) physical network

(b) virtual network composition

Fig. 5: Physical and virtual network topology for evaluation

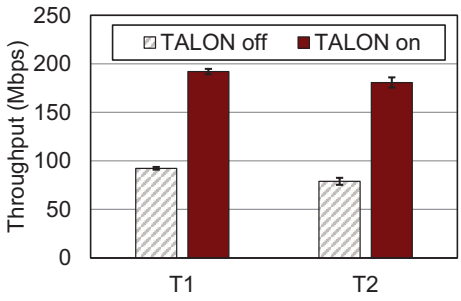


Fig. 6: Average throughput of two tenants

the throughput for the entire duration of experiment. When TALON is applied, both T1 and T2 achieve traffic throughput that is close to their requirements; however, the throughputs remain slightly under 200 Mbps because of the TCP congestion control mechanism. Regardless, the throughputs of T1 and T2 are improved 2.08 and 2.29 times, respectively.

Next, we investigate the detailed operation of TALON by measuring the throughput on each path. In this case, the traffic of T1 is generated first; 30 s hence, the traffic of T2 is generated. In addition, because we set two virtual networks with path 0, both T1 and T2 controllers generate forwarding paths using path 0. Figure 8 shows the per-path throughput during the experiment. For the first 30 s, path 0 is allocated to T1 only because the requirement for T1 (200 Mbps) can be fulfilled only by path 0. After 30 s, T2 creates its traffic and the T2 controller also calculates the forwarding path for

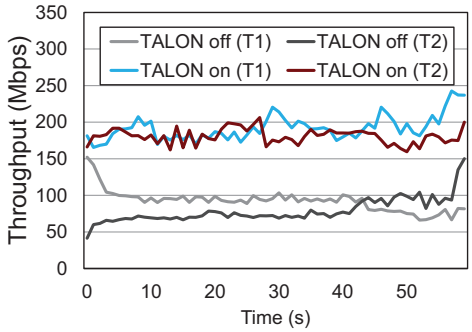


Fig. 7: Throughput of two tenants

its traffic as path 0. Given the two paths, TALON allocates throughput as follows. First, because the calculated path of T2 should be installed as the main path, a sub-path for T1 is installed (path 1). Thus, we can see that the throughput of path 1 increases after 30 s. For T2, path 2 is allocated; hence, its throughput increases after 30 s. In addition, the average throughput results achieved by tenants T1 and T2 are 209.7 and 196.5 Mbps, respectively.

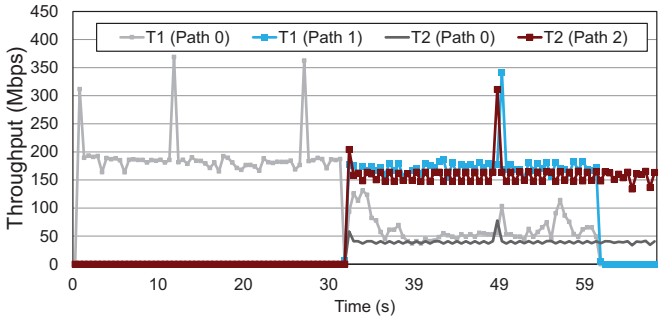


Fig. 8: Throughput variation on each tenant on the paths

V. CONCLUSION

In this paper, we present TALON, a throughput allocation scheme on vSDNs. To enable the throughput allocation in vSDN, we develop a traffic load-balancing scheme. In addition, we prioritize the main path created from the tenant over the sub-paths. We conduct experiments to demonstrate the effectiveness of TALON. The results show that throughput is improved, and traffic requirements are satisfied.

In the future, we intend to design a work-conserving allocation scheme that further improves the physical network utilization. Moreover, we expect to extend our framework to consider not only throughput but also other performance metrics including latency and reliability.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments, in advance. We also thank Anumeha for proofreading this paper. This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (No.

2015-0-00288, Research of Network Virtualization Platform and Service for SDN 2.0 Realization, and No.2015-0-00280, (SW Starlab) Next generation cloud infra-software toward the guarantee of performance and security SLA).

REFERENCES

[1] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer, "Survey on network virtualization hypervisors for software defined networking," *arXiv preprint arXiv:1506.07275*, 2015.

[2] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, vol. 1, p. 132, 2009.

[3] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Computing*, vol. 17, no. 2, pp. 20–27, 2013.

[4] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "Openvirtex: Make your virtual sdns programmable," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 25–30.

[5] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage." in *OSDI*, vol. 12. Hollywood, CA, 2012, pp. 349–362.

[6] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 1285–1293.

[7] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks." in *NSDI*, vol. 15, 2015, pp. 87–101.

[8] Y. Han, T. Vachuska, A. Al-Shabibi, J. Li, H. Huang, W. Snow, and J. W.-K. Hong, "Onvisor: Towards a scalable and flexible sdn-based network virtualization platform on onos," *International Journal of Network Management*, vol. 28, no. 2, p. e2012, 2018.

[9] K. Ko, D. Son, J. Hyun, J. Li, Y. Han, and J. W.-K. Hong, "Dynamic failover for sdn-based virtual networks," in *Network Softwarization (NetSoft), 2017 IEEE Conference on*. IEEE, 2017, pp. 1–5.

[10] W. Jeong, G. Yang, S.-M. Kim, and C. Yoo, "Efficient big link allocation scheme in virtualized software-defined networking," in *Network and Service Management (CNSM), 2017 13th International Conference on*. IEEE, 2017, pp. 1–7.

[11] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[12] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila *et al.*, "Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 1–14.

[13] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 351–362.

[14] C. Hopps, "Analysis of an equal-cost multi-path algorithm," Tech. Rep., 2000.

[15] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.