2018 IEEE 11th International Conference on Cloud Computing

FlowVirt: Flow Rule Virtualization for Dynamic Scalability of Programmable Network Virtualization

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

***Abstract*—We propose a new concept called "flow rule virtualization" (FlowVirt) for programmable network virtualization (P-NV). In P-NV, network hypervisor is a key component in that it plays a role in creating and managing virtual networks. This paper first reports a critical limitation of network hypervisor - scalability problem, which results in the high consumption of the switch memory, control channel, and CPU cycles: 3.9, 4.7, and 1.7 times higher than host-based network virtualization, respectively. This scalability problem arises because all the flow rules from the virtual network controllers are directly installed into switches. To resolve the scalability problem, FlowVirt introduces a flow rule abstraction: virtual and physical flow rules. By separating virtual and physical flow rules, the abstraction virtualizes flow rules so that FlowVirt can merge virtual flow rules to a smaller number of physical flow rules to be installed in switches. The evaluation results show the enhanced scalability of FlowVirt. The number of flow rules to be installed in switches decreases by up to 10 times compared to the previous P-NV. The control channel bandwidth and CPU cycles are also reduced by up to 14 and 3 times, respectively.**

***Keywords*-Network as a service; programmable virtual network; network virtualization; software-defined networking**

I. Introduction

In recent years, cloud computing has been widely adopted in numerous fields. For example, Netflix uses Amazon Web Services as its streaming infrastructure and Google for data backup. Similarly, various cloud-based service providers such as Apple and GE have moved their computing infrastructure to the public domain [1]. In addition, emerging carrier network architectures such as fifth-generation wireless networking systems (5G) are designed to provide diverse networking services (e.g., massive, ultra-reliable, and low-latency communication) within a single infrastructure [2]. Cloud computing is also used in edge [3] and fog [4] computing to enable such services [5], which require advanced system management for resources (e.g., IaaS). These various use cases of network resources in cloud centers are feasible with network virtualization, and the increasing number of tenants are demanding their own virtual network management functionalities (e.g., arbitrary virtual network composition, custom forwarding, and performance diagnosis for its virtual network) [6].

However, most cloud solutions provide IaaS for host nodes only, and these solutions rely on host-based network virtualization (H-NV) [7], [8], which uses overlay networking between virtual machine monitors. With H-NV, composing an arbitrary virtual network, which consists of virtual switches and links mapped to the physical topology, as well as virtual network management and optimization are restricted for each tenant [6]. In contrast, programmable network virtualization (P-NV) [9]–[11], a structure in which a network hypervisor virtualizes the entire network and switches, provides fully programmable virtual networks (VNs) and enables flexible network management for each tenant. However, P-NV has not been widely adopted so far.

In cloud networking, a key requirement is scalability [6], [12]; thus, we first investigate the amount of resources H-NV and P-NV consume when end hosts perform the same networking operations. This scalability also determines the number of tenants that can be supported in a cloud. Our investigation shows that the control channel usage and CPU cycles of P-NV are 470% and 170% higher than those of H-NV, respectively. Such poor scalability stems from the lack of abstraction of P-NV in that previously proposed network hypervisors merely install all flow rules into the physical switches in a one-to-one manner; hence, there is no abstraction for flow rules. To the best of our knowledge, we are the first to report the scalability problem of P-NV.

To address the scalability problem, we propose a new concept called flow rule virtualization (FlowVirt) that reduces the number of flow rules and thus substantially decreases the subsequent provisioning of computing resources such as the control messages and CPU cycles of the P-NV hypervisor. FlowVirt introduces an abstraction called a virtual flow rule and structures for flow rules. Thus, it separates virtual and physical flow rules to provide fine-grained virtualization. Subsequently, it maintains the mapping from virtual to physical flow rules. Using this mapping, FlowVirt merges virtual flow rules into a smaller number of physical rules.

With virtual flow rules, FlowVirt overcomes the limitations of previous flow rule compression algorithms [13]–[17]. Most compression techniques do not consider network

2159-6190/18/$31.00 ©2018 IEEE
DOI 10.1109/CLOUD.2018.00051

virtualization, and hence they suffer from high delay and complexity. Moreover, most algorithms operate proactively, i.e., a set of flow rules is provided in advance. This approach must consume additional computing resources to update the installed flow rules in compressed form. Most seriously, they cannot guarantee various network management policies. For example, some tenants might not track flows nor consider actual forwarding actions in the data plane. Even for a single tenant, the need to monitor and manage separate flow rules (e.g., load balancing and failover) may vary with the flows. In such situations, simple compression can violate tenant policies and semantics.

Therefore, FlowVirt has the following objectives:

- design a flow rule merging technique to enhance scalability;
- operate with low complexity and overhead;
- satisfy diverse flow rule management policies for tenants.

FlowVirt achieves the objectives by defining virtualization type (vt), which determines the degree of merging, and by selecting a proper vt for each virtual flow rule.

The remainder of this paper is organized as follows. Section II describes the background and motivation of this paper. The fundamental concept and design of FlowVirt are presented in Section III. Section IV explains the implementation and experimental results, and we discuss further improvements for FlowVirt in Section V. Finally, Section VI concludes the paper.

II. Background and Motivation

A. Network Virtualization Architecture

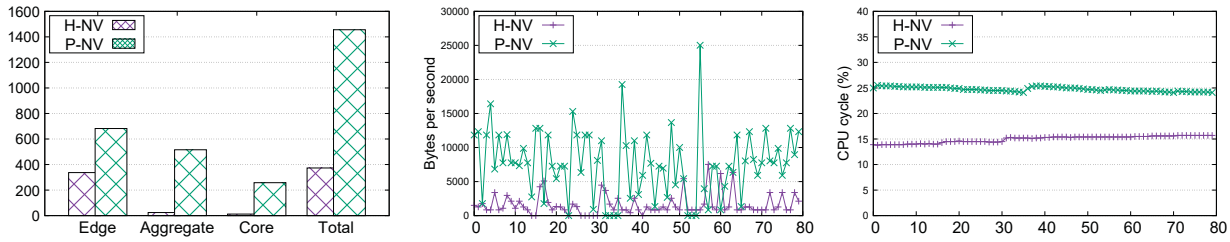
1) Host-based network virtualization (H-NV): Although overlay network virtualization techniques (e.g., VLAN, GRE, STT, MPLS, and LISP) are well suited to existing networks, cloud managers cannot manage a network in a centralized manner; thus, network management and some misconfigurations create a massive burden for cloud data centers and network operators [7]. Various methods based on software-defined networking (SDN) have been proposed to reduce this burden. For example, a network virtualization platform (NVP) [7], [18] was proposed to provide virtual networks for enterprise data centers. In such frameworks, each virtual network data path is mapped to the flow table of a software virtual switch in a host's virtual machine monitor. The policies given by a tenant are converted into the flow rule of the designated flow table of the software switch and are managed centrally by the network controller. Packets are transferred between hosts through the overlay network (tunneling). The NVP mechanism serves as the basis for NSX [19], which is a network virtualization platform for VMware. Microsoft Azure also uses host-based network virtualization. It employs a software switch called a VFP [8] that adopts the tunneling of [20] and includes other optimizations such as a unified flow table and flow table off-loading.

2) Programmable network virtualization (P-NV): Another type of NV architecture is a hypervisor-based architecture that is very similar to the system virtualization approach. Compared to H-NV, this approach permits each tenant controller to access and manage all virtual resources, e.g., virtual switches and links, directly. Thus, each tenant can completely control its virtual network. For example, a tenant can create an arbitrary virtual network topology, configure the network on demand, and achieve visibility for its virtual network, which is not possible in overlay network virtualization and H-NV. Thus far, various network hypervisors that intervene between the physical network and the controllers have been proposed. The primary objectives of P-NV are to abstract physical resources (e.g., virtual addresses and topology) into virtual ones and translate control messages between the tenant and physical network.

FlowN [10] uses VLAN for address virtualization, and the NOX controller acts as the network hypervisor. In FlowVisor [9], each tenant is given a flowspace, which is a slice of the entire address space. OpenVirteX (OVX) [11], an open-source network hypervisor, provides a variety of virtual network abstractions. OVX offers topology virtualization, including big links and big switches. It also provides address virtualization by one-to-one address allocation. Recently, ONOS implemented some core OVX functions into its design. However, none of the previous network hypervisors provide flow rule abstraction, which causes a scalability problem that we report in the next section.

B. Scalability of Programmable Network Virtualization

In P-NV, each tenant controller manages the corresponding virtual network and configures all the resources. Although P-NV provides significant benefits, this architecture has not yet been widely adopted. Because a critical criterion for cloud network infrastructure is scalability [6], [12], we investigate the scalability of P-NV in comparison with H-NV, that is widely used for network virtualization in commercial clouds nowadays. We implement H-NV and P-NV using OVX. H-NV establishes a connection between hosts by multiprotocol label switching-based tunneling [21] between edge switches, whereas P-NV provides full network configuration abilities for each tenant. For the testbed, five virtual networks (tenants) are created on a fat-tree topology (Fig. 7c), each with the same topology as the physical network. In addition, each virtual network has 5 clients and 5 servers. A client creates six TCP connections to its paired server (total of 30 TCP connections). We use three machines, which run the physical network emulator, P-NV network hypervisor (single node), and tenant controllers, respectively. The detailed machine settings are the same as those described in Section IV. Figure 1 shows the number of flow entries, control channel traffic between the network



(a) Number of flow rules (b) Control channel bandwidth (c) CPU cycle of network hypervisor

Figure 1: Scalability comparison between H-NV and P-NV

hypervisor and physical network, and the network hypervisor CPU cycles. Compared to H-NV, the resources consumed by P-NV are 3.9, 4.7, and 1.7 times higher, respectively. These differences further increase as the numbers of tenants, hosts, and applications on each host increase. The reason for the higher number of flow rules in P-NV is that the P-NV allows each tenant to create its own flow rules for virtual network. On the other hand, in H-NV, the tenant controllers do not exist nor configure the policies for themselves. The central network controller installs the flow rules in switches so that the tenant policies, e.g., flow rules to forward packets to a particular VM, are installed only at the edge switches.

The above results indicate that H-NV is more scalable than P-NV. In particular, P-NV consumes a significant amount of memory space where the flow rules are installed so that it does not fit the switch memory, e.g., ternary content-addressable memory (TCAM). Note that it has been reported that the typical size of TCAM is insufficient even in non-virtualized SDN [16], [22]. Furthermore, the massive amount of control channel traffic is a severe burden for the entire SDN because they can increase the delay of control messages. Moreover, the CPU cycle of the P-NV hypervisor is a crucial indicator for accommodating tenants because the network hypervisor is a proxy between the tenant controllers and physical network.

C. Objective

To improve P-NV scalability, we focus on reducing flow rules. When the number of flow rules increases, the hypervisor operations and the corresponding resource consumption increase. Thus, decreasing the number of flow rules is a key to improve P-NV scalability. A way to decrease the number of flow rules is compression; however, existing flow rule compression techniques generate a delay of seconds [13], [14]. In addition, most compression techniques assume proactive forwarding, which requires the flow rules between all hosts to be set up in advance. This can increase resource consumption [23]. Moreover, in the virtualized network, because the compression techniques did not consider the network management policies of tenants but just compress flow rules blindly, they are exposed to the risk of violating tenant semantics. For example, flow rules for in-network services such as firewalls cannot be compressed because the matching protocol fields and data to be matched should not be arbitrarily configured in order to guarantee the intended firewall policy. Besides, some tenants might use specific address fields for other purposes besides packet forwarding (e.g., load-balancing [24]), while others might install a forwarding path per application rather than host to differentiate the quality of services.

Hence, we design a new technique that virtualizes flow rules with a reactive forwarding scheme, which installs flow rules as needed. Moreover, we define virtualization types for specifying the degree of merging flow rules and also consider the network management policies of each tenant, which is explained in the next section.

III. Design

A. Concept and Definition

First, we introduce the key concept and mechanism of FlowVirt. Figure 2 shows how flow rules are handled with FlowVirt. In a reactive manner, the ingress switch of a new packet connection sends the flow rule request message to the network hypervisor after a new connection is generated. Then, the network hypervisor delivers the message to a tenant controller where the connection belongs. Finally, the tenant controller calculates a forwarding path for the connection and sends the composed flow rules for each virtual switch to the network hypervisor.

The previous P-NV merely delivers the flow rule from tenant controllers considering the address and topology translation. In contrast, FlowVirt defines the flow rule generated from the tenant controller as virtual flow rule (vf) and the flow rule to be implemented on the physical network as physical flow rule (pf). Each vf and pf is defined as a tuple (M, A), where $M(vf.\text{M}\ or\ pf.\text{M})$ is a matching field set and $A(vf.\text{A}\ or\ pf.\text{A})$ is an action list. We define the degree of merging called virtualization type vt. Then, FlowVirt determines an appropriate vt for each vf and virtualizes the vf into the pf, which is explained in Section III-B. FlowVirt also allows cross-tenant flow-rule mapping. After that, FlowVirt checks pf.A with installed pf.A whether the pf collides with actions of already installed pfs. If it does,

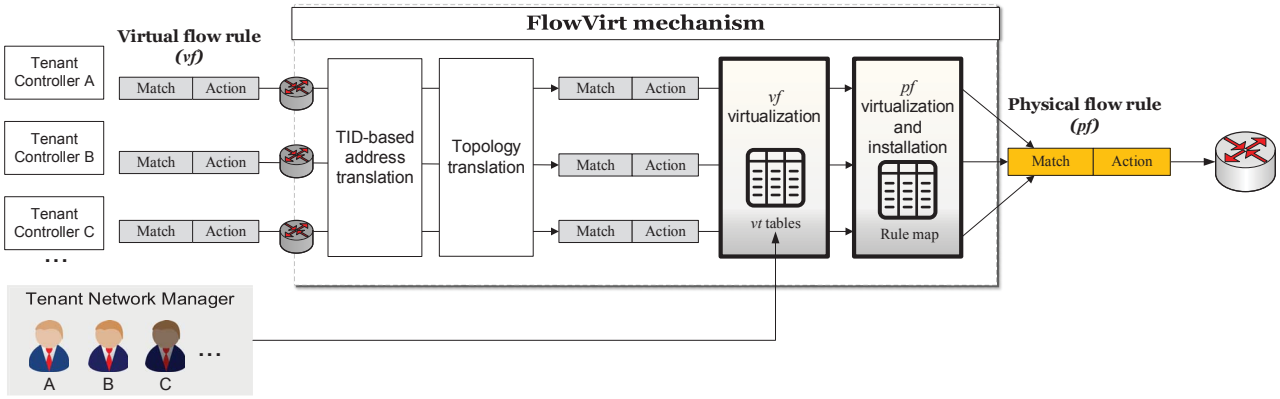


Figure 2: Flow rule processing mechanism with FlowVirt

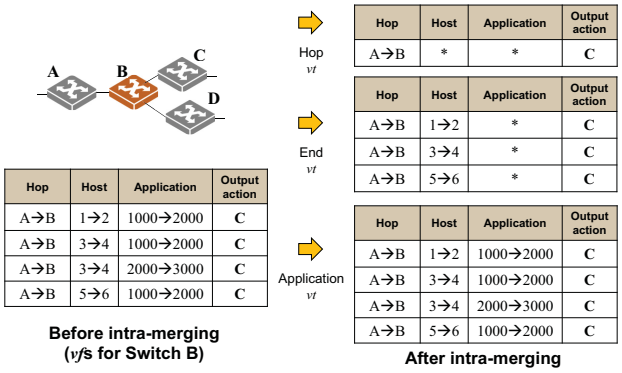


Figure 3: vf after intra-merging

Table I: Virtualization types for FlowVirt

	Inter-merging Off	Inter-merging On
Hop	hop-isolated	hop-merged
End	end	*Not available*
Application	application	*Not available*
Arbitrary	arbitrary	*Not available*

FlowVirt re-virtualizes the pf (pf virtualization in Section III-C). FlowVirt has a "rule map" that keeps the mappings of vfs and pfs (Fig. 2). Finally, FlowVirt installs the pf on the physical switch as the mapping between vf and pf.

B. vf Virtualization

Virtualization types (vt) to represent merging degrees between vf and pf are of two types: 1) merging within one tenant, which we refer to as "intra-merging," and 2) merging between tenants, called "inter-merging."

First, intra-merging varies with the granularity of the matching unit – the number of flow rules merged. For instance, when a flow rule matches an application pair–host address and application address (e.g., port number), the flow rule controls only the application identified with the address; thus, each application is processed by a single flow rule. On the other hand, if the flow rule matches the host address pair (of end hosts) without an application address pair, all applications (with any port number) between the end hosts are processed by that rule.

Therefore, FlowVirt defines four vts of intra-merging: hop-based (hop), end-host-based (end), application-based (application), and without intra-merging (arbitrary). The "hop" vt creates a pf that matches only the hop address of vf. "End" vt creates a pf that matches up to the hop and host addresses of vf, and "application" vt creates a pf that matches the hop, host, and application addresses. "Arbitrary" vt is the same as in the previous P-NV (no merging). Suppose there are switches A, B, C, and D (Fig. 3), where traffic flows from switches A to B, and the flow table of switch B, which consists of four rules (the left table of Fig. 3). The flow rules are merged by vt to the right tables of Fig. 3: "hop," "end," and "application" vt.

Second, inter-merging between tenants is also possible. Among the methods proposed for address translation in P-NV, FlowVirt uses tenant identifier (TID) allocation, which makes each packet contain TID [10], [25]. If the tenants are different, FlowVirt achieves inter-merging by not matching the TID.

Based on intra- and inter-merging, there are eight possible vts. The "hop" vt can be used with inter-merging because the physical hop addresses are not affected by TID. Thus, we have "hop-isolated" vt and "hop-merged" vt, respectively (Table I). However, regarding "end" and "application" vts, inter-merging is not allowed. This is because each host and application belong to a single virtual network with an "end" or "application" vt so that FlowVirt should resolve the overlapping virtual addresses between tenants (address translation), which is not possible. As a result, FlowVirt defines "end" and "application" vt without inter-merging. Furthermore, with "arbitrary" degree, FlowVirt cannot be certain of how a tenant controller composes the vf. Thus,

Flow information example

Switch ID	Ingress port	Ethernet source	Ethernet destination	Ethernet type	IP source	IP destination	IP protocol	Source port	Destination port	Virtualization type	Priority	Range
*	*	*	*	*	*	*	*	*	*	Hop-merged	Low	⇨ Entire virtual network
4	*	*	*	*	*	*	*	*	*	Hop-isolated	High	⇨ Specific virtual switch
*	*	*	*	*	10.0.0.1	10.0.0.2	*	*	*	End	Mid	⇨ Particular hosts
*	*	*	*	*	*	*	*	*	12345	Application	Mid	⇨ Particular application

Figure 4: vt table and vt policy example

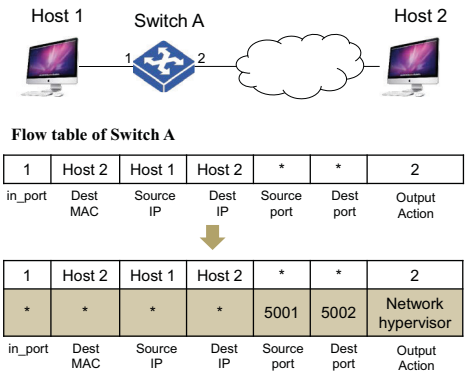


Figure 5: Indicator rule installation for "application" vt policy

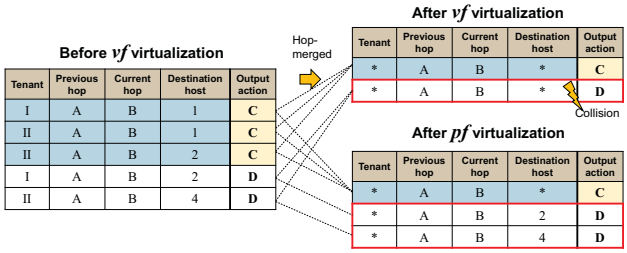


Figure 6: pf virtualization for "hop-merged" vf

Table II: Packet header semantics of FlowVirt

Address fields	Packet header semantics
in_port	Previous hop address
Source MAC	Tenant identifier
Destination MAC	Next hop address
Source IP	Source end host address
Destination IP	Destination end host address
Source Port	Source application identifier
Destination Port	Destination application identifier

an "arbitrary" vt cannot merge flow rules across tenants. Hence, we have five vt for FlowVirt: "hop-merged," "hop-isolated," "end," "application," and "arbitrary" (Table I).

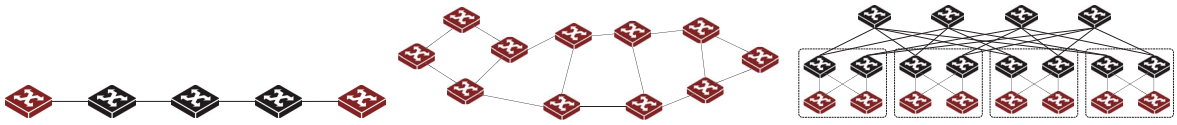
Now, FlowVirt should select a vt to be applied for each vf. Because each tenant has its own policy for managing the virtual network, the selection depends on the network management policy of the tenant. Hence, FlowVirt gets the vt policies from the tenants and puts them in the vt table (Fig. 4). Each entry of the vt table is a vt policy, and FlowVirt allocates one table per tenant. The syntax of a vt policy is as follows: switch ID, flow information, vt, and priority. Using the syntax, a tenant can specify the vt policy for its virtual network, specific virtual switch, particular hosts, or a particular application. For example, the first entry in Fig. 4 shows that all of the vfs in the tenant are applied as "hop-merged" because the entry matches any switch, hosts, and applications. In addition, the second entry shows the vt policy for a specific switch by designating one virtual switch (switch ID 4 in Fig. 4) and not defining any flow information. The third and last entries show vt policies for specific hosts and applications by specifying the hosts and applications in the flow information in Fig. 4. Moreover, a tenant can give priority between vts to provide one vt policy when multiple vt policies are applicable for a vf.

Using the vt table, the selection of a vt for each vf is done as follows. FlowVirt looks up vt table for each flow rule by matching the virtual switch ID to the switch ID and the vf.M to the flow information. Out of the matched vt table entries, the vt with the highest priority is selected.

When vt table has a vt policy for "application" vt, FlowVirt installs an "indicator rule." In SDN, the first switch reports a flow rule request to the network hypervisor when the switch does not have any rules to process a new packet. However, when a pf that matches packets in a coarser-grained manner than "application" (such as "end," "hop-merged," and "hop-isolated") is already installed in the physical switch, a finer-grained flow rule cannot be installed since the application packet is processed with the existing coarser-grained pf. Hence, the request for the pf generation cannot be created. To illustrate this situation in detail, assume that the vt table has an "end" vt with low priority, and an "application" vt for traffic from ports 5001 to 5002 with higher priority (Fig. 5). If the flow table of switch A, which is the first switch of the path between hosts 1 and 2, already has a pf virtualized with the "end" vt, the first packet from application ports 5001 to 5002 will be

Table III: Matching protocol fields of converted pf.M in FlowVirt implementation

Virtualization type	Ingress switch	Egress switch	Core switch
Hop-merged	in_port, MAC pair	in_port, source MAC, destination IP	in_port, destination MAC
Hop-isolation	in_port, MAC pair	in_port, source MAC, destination IP	in_port, MAC pair
End	in_port, MAC pair, IP pair		
Arbitrary	in_port, MAC pair, IP pair, port pair		
One-to-one	Include in_port and source MAC address		



(a) Linear topology (b) Internet2 topology (c) Fat-tree topology

Figure 7: Topology

forwarded by this flow rule, not by the pf that is virtualized with the "application" vt. Thus, a proper pf (from ports 5001 to 5002) is not created. This problem happens because the flow rule to match with the coarse-grained flow rule is already installed in the physical switch. To solve this problem, FlowVirt creates an indicator rule, which forwards the first new application packet to the network hypervisor in a higher priority than the previously installed pf. Then, FlowVirt creates a new pf that is virtualized in "application" vt (5, lower table) so that subsequent packets of "application" vt are handled properly. In addition, this problem can occur similarly between "end" and "hop-merged" or "hop-isolated" vt policies. However, in packet forwarding, the flow rule in the first switch (ingress flow rule) should match the host address because the previous hop of an ingress switch is the source host. In other words, the ingress flow rule is always at least in the "end" vt. Hence, FlowVirt only needs to install indicator rule only for the "application" vt policy, not for the "end" vt policy.

C. pf Virtualization and Installation

After creating a pf, FlowVirt checks pf.A to determine whether pf.A collides with already installed pfs in the physical switch. Suppose the topology in Fig. 3, where traffic flows from switches A to B, and the flow table of switch B consists of five rules (the left table of Fig. 6). Assume that flow rules are sequentially generated from the top to the bottom of the left table in Fig. 6. When the vfs are all virtualized with "hop-merged," only the hop addresses are matched as in the right upper table in Fig. 6. If the top three flow rules are virtualized and installed on the physical switch, the packets from hops A to B can be forwarded to C. However, the fourth flow rule matches the same hop, but sends the packet to D, which is a collision with the already installed pf. To deal with this problem, FlowVirt performs pf virtualization that virtualizes the collision pf with the "end" vt using the existing pf information in the rule map.

Table IV: vts of experiment cases

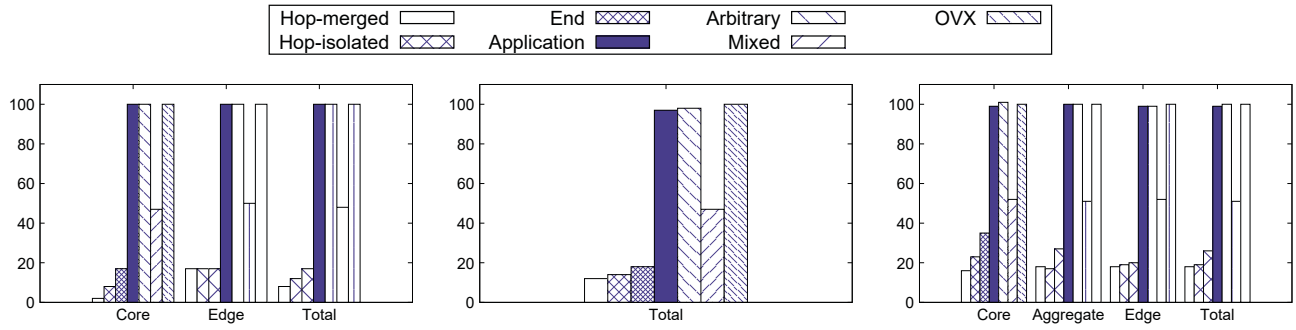
Case	Tenant1	Tenant2	Tenant3	Tenant4	Tenant5
Hop-merged	hop-merged				
Hop-isolated	hop-isolated				
End	end				
Application	application				
Arbitrary	arbitrary				
Mixed	hop-merged	hop-isolated	end	application	arbitrary
OVX	*Not available*				

The right lower table of Fig. 6 shows the results of pf virtualization. This collision can occur for the pf virtualized by "hop-isolated" vt as well. Thus, FlowVirt checks this collision and performs pf virtualization for pf virtualized by "hop-merged," and "hop-isolated" vts.

After pf virtualization, FlowVirt installs the pf into the physical switch. If the pf already exists in the switch, FlowVirt maps the vf to the existing pf. If not, FlowVirt installs the flow rule and updates the pf on the rule map.

IV. Implementation and Evaluation

We implement FlowVirt based on OVX [11] with OpenFlow. For address translation, we use the scheme proposed in [25] and implement an additional routine to ensure that each packet and the flow rules in the physical network have the same packet header semantics for TCP/IP (Table II). The in_port is the matching field for the previous hop, and source MAC and destination MAC is for TID and next hop address, respectively. In addition, IP, port pairs are addresses for the host and application pair. Moreover, the FlowVirt implementation converts vf.M upon vt, as shown in Table III. We believe that southbound interface protocols (e.g., P4



(a) Linear topology (b) Internet2 topology (c) Fat-tree topology

Figure 8: Comparison of the number of flow rules (%)

[26]) other than OpenFlow can be used because FlowVirt only depends on the packet field matching of the protocol.

Next, we investigate the performance and overhead of FlowVirt. The results are compared with those obtained to analyze the scalability of P-NV solutions with and without FlowVirt. We use Mininet [27] as the SDN network emulation testbed. In addition, ONOS is used for the virtual network controller. Mininet, FlowVirt, and ONOS controllers run on separate machines connected to each other via a 10-GbE switch. One ONOS instance is provided for each tenant, and all instances are executed using Docker within the single machine. We set the controllers to provide flow rules that match hop, host, and application address pairs. In this study, FlowVirt is evaluated on a software-emulated physical network, rather than actual cloud infrastructure. Although a software-emulated network is not the real cloud infrastructure, we believe that our experiments exhibit the main characteristics of FlowVirt. Further evaluation of the physical testbed with real traffic remains as future work.

To evaluate FlowVirt under various conditions, we use three topologies. Figure 7a shows a linear topology (the simplest case). In addition, Fig. 7b, a simplified model of the Internet2 NDDI testbed, is similar to the topology evaluated in [11]. We use the topology to simulate networks where the switches perform both edge and core switch roles. Figure 7c shows a fat-tree topology, which is typical for a data center. For all topologies, five virtual networks were created with the same switches and links of the physical network and two hosts per edge switch. In the linear topology, the switches at the end of both sides act as edge switches. The deepest level switches are the edge switches for the fat-tree topology. All switches are edge switches in the Internet2 topology. We used iperf3 to generate traffic. Each host in a virtual network acts as either a client or a server. The client generates six separate TCP connections to its paired server for 60 s, and all the traffic is generated at intervals of 1 s. Furthermore, to show the dynamic merging degree of FlowVirt, we conduct each experiment on seven cases varying vt for each tenant. The experiment cases are summarized in Table IV.

A. Number of Flow Rules

We evaluated the resource usage enhancement of FlowVirt (compared to OVX) for the number of flow rules. Figure 8 shows the ratio of the total number of flow rules of each switch to the number of OVX flow rules. The "hop-merged" vt in the linear topology reduces the number of flow rules in OVX by up to 91% (10 times flow rule reduction). In contrast, the fat-tree and Internet2 topologies reduce these numbers by up to 82% and 87%, respectively. In the linear topology, there is only one path for all traffic. Thus, these results show the maximum flow rule merging power. In contrast, for the fat-tree and Internet2 topologies, various paths with the same costs are possible for each traffic flow. Thus, the degree of merging decreases accordingly. However, FlowVirt still achieves a flow rule reduction of five times.

Between the "hop-merged" and "hop-isolated" cases, "hop-merged" merges the flow rules by ignoring the tenants. Therefore, the "hop-merged" case reduces the number obtained by the "hop-isolated" case by 80%. For the "hop-isolated" and "end" cases, the "hop-isolated" case halves this number because the flow rules of two hosts in the "end" case are merged into a single rule. Of the experiment cases, "application," "arbitrary," and "OVX" show nearly the same number of flow rules. Because the tenant creates the vf that matches up to application addresses in our setting, the number of pf created for "application" vt is the same as the number of vfs. Besides, "arbitrary," which does not perform merging, yields a similar number of flow rules as "OVX." The core flow rules in the linear topology follow this tendency. However, the other topologies show a weaker effect because a single traffic flow can have multiple paths with the same cost, and each tenant controller arbitrarily selects one of these paths.

B. Computing Resource usage

1) Control channel bandwidth: The control channel connects the network hypervisor and physical network. Table V summarizes its usage relative to the average number and

Table V: Control channel usage

(a) Average number of FlowMod messages (packets per second)

Topology	Hop-merged	Hop-isolated	End	Application	Arbitrary	Mixed	OVX
Linear	1.82	2.68	5.36	23.39	24.46	11.32	22.00
Internet2	2.82	2.99	5.94	30.29	26.96	13.33	28.69
Fat-tree	4.07	4.52	10.43	37.49	34.16	20.07	37.39

(b) Bandwidth usage for FlowMod messages (bytes per second)

Topology	Hop-merged	Hop-isolated	End	Application	Arbitrary	Mixed	OVX
Linear	321	460	902	3939	3650	1839	4050
Internet2	485	512	1003	5131	4037	2152	5314
Fat-tree	689	761	1749	6317	5093	3227	6914

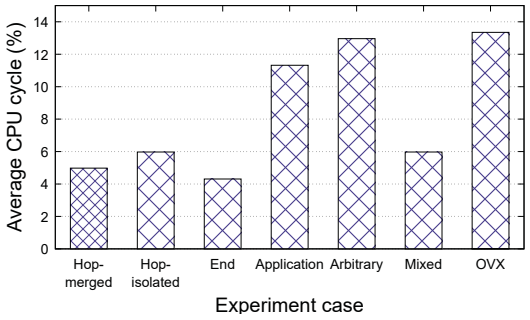


Figure 9: CPU usage

size of flow rule modification (FlowMod) packets. First, the FlowMod message numbers show that the number of requests (Table Va) for a new flow decreases as the flow rule matches the coarse-grained addresses. In addition, "application," "arbitrary," and "OVX" create a similar number of FlowMod messages because they all install the same number of pfs in the physical network in our setting.

Table Vb shows that OVX consumes the highest bandwidth among seven cases. This is because the flow rules contain additional actions for address translation, which is to rewrite a MAC address pair, and edge flow rules also have additional actions for rewriting IP pairs. On the other hand, FlowVirt requires the action for edge flow rules only, and so the size of the control message for the action is smaller than OVX because FlowVirt just rewrites the source MAC field. FlowVirt adds actions that rewrite MAC address pair to enable "hop-merged" and "hop-isolated' vt, but the number of additional actions is still smaller than OVX. Furthermore, even for the "arbitrary" case where FlowVirt installs the highest number of flow rules, the bandwidth for the message is lower than OVX.

2) CPU cycle of network hypervisor: Figure 9 shows the average CPU usage. The CPU cycles of "hop-merged," "hop-isolated," and "end" are quite similar. This is because the numbers of flow rules of them are similar. Likewise, the CPU cycles of "application," "arbitrary," and "OVX" are alike.

Table VI: Delay for flow rule virtualization (ms).

	Delay	Increased delay
OVX	5.05	-
Hop-merged	5.74	0.69
Hop-isolated	5.90	0.84
End	5.86	0.81
Application	5.86	0.80
Arbitrary	5.84	0.79
Mixed	5.98	0.92

C. Overhead

Next, we evaluate the overhead of FlowVirt compared to OVX. We use the linear topology with seven experiment cases (Section IV). The metric for overhead is the time required to virtualize a flow rule, that is, the additional delay to modify a FlowMod message. Table VI shows the results. The average increased delay is approximately 0.8 ms, which is about 16% over OVX. Since the increased delay occurs only once during flow rule installation, we believe that FlowVirt does not have a significant effect on the overall packet forwarding and processing.

V. Discussion

A. Isolation of Flow Rules

The arbitrary merging between vfs may break isolation between tenants. To maintain isolation, FlowVirt uses a vt table for each tenant, with which each tenant can specify its vt policies for vfs. Therefore, the merging between vfs is performed only when vt permits.

B. Extension to One-to-many Mapping between vf and pf

In this study, we only consider how to maintain many-to-one mapping between vf and pf. However, there is also a need for one-to-many mapping. For example, the network hypervisor can support traffic load balancing by mapping a vf to two pfs, which splits traffic into two paths. Because FlowVirt separates vf and pf and the mapping is decided by vt, FlowVirt can be extended to support one-to-to-many mapping by introducing new vt. We leave this for our future

work to improve virtual network performance via traffic engineering.

VI. Conclusion

SDN-based programmable network virtualization is a promising technology that accommodates diverse services within the cloud infrastructure. However, the scalability limitation of the P-NV is critical. This study attempts to resolve this problem with FlowVirt, which increases the scalability of the P-NV by virtualizing flow rules. To the best of our knowledge, this is the first attempt to virtualize flow rules by mapping virtual and physical flow rules to merge "similar" flow rules. When virtualizing flow rules, FlowVirt takes the tenant's policy into consideration, which is not to break isolation between tenants and also to provide proper network management for each tenant.

Our evaluation results show that the number of flow rules and the control channel bandwidth decrease by up to 10 times compared to OVX. In contrast, although FlowVirt consumes additional CPU cycles for flow rule virtualization, it saves the overall CPU usage because the number of virtual rules is reduced. Thus, the overall CPU usage of FlowVirt is lower by up to 32% than that of OVX.

In the future, we plan to extend FlowVirt to support one-to-many mapping between pfs and a vf to provide multiple paths for a virtual path. This mapping enables the network hypervisor to support multipath-based traffic engineering.

Acknowledgment

We thank the anonymous reviewers for their valuable and insightful comments. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2015-0-00288, Research of Network Virtualization Platform and Service for SDN 2.0 Realization). This research was also supported by the MSIT (Ministry of Science and ICT), Korea, under the SW Starlab support program (2015-0-00280) supervised by the IITP.

References

[1] J. Weinman, "Migrating to–or away from–the public cloud," *IEEE Cloud Computing*, vol. 3, no. 2, pp. 6–10, 2016.

[2] I. F. Akyildiz, S. Nie, S.-C. Lin, and M. Chandrasekaran, "5g roadmap: 10 key enabling technologies," *Computer Networks*, vol. 106, pp. 17–48, 2016.

[3] R. S. Montero, E. Rojas, A. A. Carrillo, and I. M. Llorente, "Extending the cloud to the network edge." *IEEE Computer*, vol. 50, no. 4, pp. 91–95, 2017.

[4] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[5] L. Gupta, R. Jain, and H. A. Chan, "Mobile edge computing-an important ingredient of 5g networks," *IEEE Software Defined Networks Newsletter*, 2016.

[6] P. Costa, M. Migliavacca, P. R. Pietzuch, and A. L. Wolf, "Naas: Network-as-a-service in the cloud." in *Hot-ICE*, 2012.

[7] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. J. Jackson *et al.*, "Network virtualization in multi-tenant datacenters." in *NSDI*, vol. 14, 2014, pp. 203–216.

[8] D. Firestone, "Vfp: A virtual switch platform for host sdn in the public cloud." in *NSDI*, vol. 17, 2017, pp. 315–328.

[9] 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.

[10] 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.

[11] 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.

[12] R. Buyya, S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. Netto *et al.*, "A manifesto for future generation cloud computing: Research directions for the next decade," *arXiv preprint arXiv:1711.09123*, 2017.

[13] A. X. Liu, C. R. Meiners, and E. Torng, "Tcam razor: A systematic approach towards minimizing packet classifiers in tcams," *IEEE/ACM Transactions on Networking (TON)*, vol. 18, no. 2, pp. 490–500, 2010.

[14] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: A non-prefix approach to compressing packet classifiers in tcams," *IEEE/ACM Transactions on Networking (ToN)*, vol. 20, no. 2, pp. 488–500, 2012.

[15] S. Banerjee and K. Kannan, "Tag-in-tag: Efficient flow table management in sdn switches," in *Network and Service Management (CNSM), 2014 10th International Conference on*. IEEE, 2014, pp. 109–117.

[16] K. Kannan and S. Banerjee, "Compact tcam: Flow entry compaction in tcam for power aware sdn," in *International Conference on Distributed Computing and Networking*. Springer, 2013, pp. 439–444.

[17] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan, "Scalable rule management for data centers." in *NSDI*, vol. 13, 2013, pp. 157–170.

[18] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the network forwarding plane," in *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*. ACM, 2010, p. 8.

[19] N. VMware, "Network virtualization platform."

[20] 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," in *ACM SIGCOMM computer communication review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[21] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture," Tech. Rep., 2000.

[22] Z. Teo, K. Birman, and R. Van Renesse, "Experience with 3 sdn controllers in an enterprise setting," in *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 2016, pp. 97–104.

[23] C. Yu, C. Lumezanu, H. V. Madhyastha, and G. Jiang, "Characterizing rule compression mechanisms in software-defined networks," in *International Conference on Passive and Active Network Measurement*. Springer, 2016, pp. 302–315.

[24] N. Kang, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on sdn switches," *Tech. Rep.*, 2015.

[25] B.-y. Yu, G. Yang, K. Lee, and C. Yoo, "Aggflow: Scalable and efficient network address virtualization on software defined networking," in *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*. ACM, 2016, pp. 1–6.

[26] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[27] 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.