LiteVisor: A Network Hypervisor to Support Flow Aggregation and Seamless Network Reconfiguration for VM Migration in Virtualized Software-defined Networks

GYEONGSIK YANG1, BONG-YEOL YU1, SEONG-MUN KIM2, AND CHUCK YOO.1 (Member, IEEE)

1Department of Computer Science and Engineering, Korea University, Seoul 02841, Republic of Korea
2KT R&D Center, Seoul 06763, Republic of Korea

Corresponding author: Chuck Yoo (e-mail: chuckyoo@os.korea.ac.kr).

This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korean 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 (Institute for Information & Communications Technology Promotion).

ABSTRACT Network virtualization based on software-defined networking (SDN) has become a necessary technology to provide various services in cloud datacenters. Although many network hypervisors have been proposed to support SDN-based network virtualization, their forwarding techniques excessively consume the limited ternary contents addressable memory (TCAM) of OpenFlow-enabled switches. Moreover, they do not consider network reconfiguration after virtual machine migration. In this paper, we propose LiteVisor, which resolves the two problems mentioned above. It develops the Locator, Identifier, and Tenant sEparating (LITE) scheme. LITE-based forwarding enables flow aggregation which reduces switch memory consumption. In addition, LiteVisor provides seamless network reconfiguration which does not require tenant controllers to be aware of virtual machine migration based on LITE. We evaluate LiteVisor in terms of the number of flow table entries and network reconfiguration time and compare it with OpenVirteX, which is an open-source network hypervisor. The results show that the number of flow rules decreases by up to eight times compared to OpenVirteX in a fat-tree topology. We also demonstrate the seamless network reconfiguration of LiteVisor in the experiments and present the results of the reconfiguration time by the topology size and packet sending interval of hosts.

INDEX TERMS Computer network management, Network reconfiguration, Network virtualization, Software defined networking, VM migration

I. INTRODUCTION

DATA centers are widely deploying network virtualization to enable communications between virtual machines (VMs) and servers [1], and they are now composing their networking infrastructure with hardware switches capable of SDN management [2]. Software-defined networking (SDN) solutions are especially prevalent, as SDN-based network virtualization allows each tenant to flexibly create an arbitrary virtual network topology and implement its own network policies (flow rules), which enables proper provisioning for various services within a single datacenter infrastructure. The core component of SDN-based network virtualization is a network hypervisor, such as FlowVisor [3], FlowN [4], OpenVirteX [5] (OVX), and others [6]. A network hypervisor intervenes between virtual network controllers (tenant controllers), such as ONOS, OpenDayLight, or Floodlight, and the physical SDN (pSDN). To allow full control of a virtual SDN (vSDN) per tenant, a network hypervisor isolates flows and translates control messages from tenant controllers, and vice versa. Thus, each tenant is granted considerable...
autonomy to set up packet flows in the virtual network and can control the entire virtual switches.

Network hypervisors require the use of a specific forwarding technique to isolate vSDNs. For example, FlowN forwards packets with virtual local area networks (VLANs) [7] to separate flows by an attached tenant identifier (TID). On the other hand, OVX, which uses its own forwarding technique, assigns physical IP (pIP) and physical MAC (pMAC) addresses and configures the flow table with these pIPs and pMACs. Virtual IPs (vIPs) and MACs (vMACs) are translated to pIPs and pMACs at the edge switches of vSDNs, so each virtual network flow is separated from others. Flow rules from the network hypervisor are stored in limited memory for fast lookup, i.e., ternary contents addressable memory (TCAM) [8]. Existing forwarding schemes overflow this memory easily because the flow rules from vSDNs are translated into the physical network and installed in the switch without considering the memory limitations of the switch. Thus, to reduce the switch memory consumption in vSDN, we propose a novel forwarding technique that reduces the number of necessary flow rules, improving the capacity and performance of vSDN systems.

Moreover, in datacenters, the mobility of a VM is an essential attribute to enhance the utilization of physical infrastructure and achieve the performance demanded by users [9]. However, previous network hypervisors usually assumed the rigid location of hosts and did not provide any seamless network reconfiguration techniques; thus, they are prone to malfunction, and user services are disrupted when the hosts are relocated. This is because the network hypervisors are responsible for physical network configuration but do not support the movement of VMs.

To overcome the problems mentioned above, we propose LiteVisor for flow aggregation and seamless network reconfiguration. LiteVisor introduces the Locator, Identifier, and Tenant Separating (LITE) concept and LITE-based forwarding scheme (LForwarding). LForwarding enables LiteVisor to aggregate flows belonging to the same tenant and host locations (ingress and egress edge switches) by encapsulating packets with their tenant-separated locators. Thus, the number of flow rule entries in intermediate physical switches (pSwitches) can be reduced considerably. In addition, by managing the identifiers and locators of VMs separately in the network hypervisor, LForwarding effectively supports network reconfiguration and hides the migration of a VM from the tenant controller so that each tenant can choose its network controller without concerning the VM migration support of its controller.

Our work achieves the following contributions.

- We present LiteVisor, a network hypervisor that aggregates flows eight times better than the existing ones.
- We develop a network reconfiguration technique to support VM migration efficiently.

1 The words ‘host’ and ‘VM’ are used interchangeably throughout the paper.

- LiteVisor is implemented based on an open-source network hypervisor and evaluated with various network topologies.

The remainder of the paper is organized as follows. Background, related work, and motivation are summarized in Section II. Section III illustrates the architecture of LiteVisor and explains the designs in detail. In Section IV, we present the results of experiments conducted to evaluate the performance, and Section V discusses the limitations and future work of LiteVisor. Finally, Section VI concludes this paper.

II. BACKGROUND AND MOTIVATION

A. SDN-BASED NETWORK VIRTUALIZATION

Network virtualization is a popular research topic in networking, in that it enables the consolidation of physical resources among various networking services [10]–[14]. However, the rigidity of TCP/IP networking architecture has led to challenges in the control of each virtual network. To address the rigidity associated with the physical architecture, SDN has emerged to enable new networking technologies to be prototyped into existing network infrastructures by decoupling the control and forwarding functions of the network elements. Then, by centralizing the control functions, perfect control for each virtual network has been made possible [15]. With this in mind, various studies have focused on the virtualization of network resources, such as switches, addresses, and policies.

Figure 1 depicts the architecture of SDN-based network virtualization. The network hypervisor abstracts physical switches and provides a virtual network composed of arbitrary switches and links in the physical network. Each tenant operates its own tenant controller in a separate VM or machine in a datacenter, e.g., POX, Floodlight, ONOS, or OpenDayLight. Through the network hypervisor, the tenant controller manages switches in its network. For physical switches, the network hypervisor becomes a controller by delivering and translating control messages from tenant controllers and vice versa. The placement and form of tenant controllers and network hypervisors have been surveyed in various literatures [16]–[18].
FlowVisor [3], an initial SDN network hypervisor, focuses on isolating experimental network traffic from production network traffic by using the OpenFlow (OF) protocol [19]. As FlowVisor is located between the tenant controller and the pSwitch, it controls the views of the tenant controller on the pSwitch. Moreover, FlowVisor assigns a subspace of the header field space, which is a flow space, to each vSDN, such that the flow spaces of unique vSDNs do not overlap. FlowVisor also provides the vSDN with isolation for bandwidth, topology, switch CPU, flow space, and OF control channel. However, there exists a limitation in that the tenants using FlowVisor cannot use the full flowspace. They can only use the sliced flowspace that is allocated to each of them specifically, and if they try to use other regions, tenants receive an error message.

FlowN [4] is another network hypervisor for virtualizing SDN networks. FlowN completely abstracts the pSDN topology to the tenants instead of partitioning the pSDN. Moreover, the tenant is decoupled from resource management because FlowN presents the virtual address space to the tenant. Each vSDN can use an arbitrary virtual address on its own; thus, FlowN maps virtual address space to the physical address space. In the pSDN, all packets are encapsulated with the VLAN, and the VLAN ID becomes the TID. Thus, FlowN virtualizes the addresses by matching both the TID and virtual addresses. However, tenants are not permitted to use their own vSDN controller. Instead, FlowN uses containers to host tenant controllers upon the NOX controller [20]; thus, the FlowN user is restricted to the NOX controller. This means that FlowN does not exploit the advantage of an SDN network hypervisor that includes the ability to adopt heterogeneous controllers.

On the other hand, OpenVirteX (OVX) [5], which is an open-source network hypervisor, allows each tenant to deploy any controller and provides an entirely virtualized network for each tenant. A key component of OVX is address virtualization. OVX denotes the arbitrary allocated addresses for a host as virtual addresses, such as virtual IP (vIP) and virtual MAC (vMAC). To resolve the overlapping of virtual addresses between tenants, OVX assigns new physical addresses for virtual addresses of each host. For a vIP, a physical IP (pIP) is allocated as a combination of a TID (in the upper 8 bits of the IP) and a host identifier in each tenant (in the lower 24 bits of the IP). Similarly, OVX assigns a pMAC to each vMAC. Then, OVX provides arbitrary address selection for each tenant by translating control messages, i.e., modifies OF messages containing virtual addresses (such as flow rule installation) into new OF messages with allocated physical addresses.

Suppose that Host1 sends packets to Host2 (Figure 2(a)). The flow table of the ingress pSwitch (InpSwitch) connected to Host1 is set up with the newly assigned pIP and pMAC of Host1 and Host2, as shown in Figure 2(b). When packets reach the InpSwitch, it translates vIPs and vMACs of Host1 and Host2 into their pIPs and pMACs via action, respectively. The intermediate pSwitch (InterpSwitch) forwards packets based on pIPs and pMACs as shown in Figure 2(c). The egress pSwitch (EpSwitch) connected to Host2 retranslates pIPs and pMACs to vIPs and vMACs in order to forward packets to Host2, as shown in Figure 2(d). When the packets flow in bi-direction, the forwarding procedure is performed with two rules per switch, one for Host1 to Host2 and another

![Connection from Host1 to Host2](image)

**FIGURE 2.** Forwarding scheme of OpenVirteX
for the opposite direction.

In another study, [21] proposed a compositional hypervisor that combines flow rules from multiple controllers. The purpose of this hypervisor is to manage a single network with heterogeneous SDN controllers with various functions. The hypervisor provides a combination of multiple flow rules, created by controllers, with parallel and sequential operators. This reduces the number of flow rules, but the hypervisor is not targeted towards operating multiple virtual networks which can be used for each tenant. The network configuration of each tenant controller is not isolated, but is instead combined with others such that they operate as a single physical network. This is different from our target environment in which multiple controllers operate their virtual network with network hypervisor’s abstractions.

ONVisor [22] is another hypervisor that resides in the ONOS controller. This hypervisor contains similar network abstractions and flow rule processing mechanisms as compared to OVX; however, ONVisor implements a distributed network hypervisor architecture based on the ONOS controller and proposes a controller-hypervisor combined architecture.

### B. SHORTAGE OF FLOW RULE MEMORY

The forwarding schemes of SDN-based network hypervisors (e.g., OVX in Figure 2(a)) simply translate the flow rules created from the tenant controllers according to policies which reflect the address and topology virtualization. These methods are inefficient in terms of switch memory consumption because all the flow rules are installed for each connection even if the connections are transmitted on the same flow path.

Most switches store flow tables in specialized hardware, i.e., TCAM, for fast flow rule searching [23]. However, according to Yu et al. [24], typical top-of-rack forwarding table in data centers require 78K rules; however, this number is larger than what most OF switches provide by the order of magnitude. Moreover, the flow rules in SDN consume much more memory space than traditional networking because SDN allows a flow rule to be flexibly composed. In traditional networking architecture, packet matching requires two to five tuples (i.e., address and protocol type fields). However, due to the programmable design of SDN, each switch must match packet fields as many as seven times (i.e., classifier-id, protocol flags, the port number of switches, and security tags), compared to general L2 switches. [25], [26].

Therefore, techniques for reducing memory consumption have been widely studied. The scope of rule aggregation and the way of aggregation is different from study to study. Liu et al. [27] proposed a prefix-based flow rule compression algorithm. In addition, Bit weaving [25] views flow rules as bit-strings and creates the compressed prefix classifier for pre-given flow rules. Their algorithm swaps and merges bits of the strings to enhance the compression power of the previous study [27]. Although both approaches are well-known studies for flow rule compression, these algorithms have high complexity and incur significant delays (e.g., seconds) in compressing flow rules in the pSwitches. Because network hypervisors also contribute to the delay in translating each control message to each virtual network for isolating resources, the increased delay is more critical to vSDN.

Alternatively, segment routing [28] is a kind of source routing using two labels, which are referred to segments. To forward packets, a node segment for identifying the switch and an adjacency segment that designates the output link within the switch are used. An ingress switch or source host encodes the segment list (forwarding path) with multiple MPLS tunneling for each packet, and all switches contain flow entries for matching node segments. The switch also has flow entries for matching an adjacency segment as the number of the ports each switch has. Then, each switch has flow entries for the number of all switches in the network, including the number of ports of each switch.

JumpFlow [29] proposed a forwarding scheme which is similar to source routing to reduce flow rule consumption. JumpFlow encodes the forwarding path using VLAN ID. Because the VLAN ID space is limited, JumpFlow divides the forwarding path information and proposes a placement algorithm for flow rules which update the VLAN ID.

STAR [30] proposed an admission control technique and routing scheme to avoid flow table overflow. STAR first checks the availability between end-hosts’ communication by maintaining flow table loads and path-sets in controllers and checking the full of loads of available paths between end-hosts. In addition, to calculate a path for a new connection, STAR uses a path cost, considering both the flow table utilization and path length.

Segment routing, JumpFlow, and STAR differ from LiteVisor in three aspects. First, in terms of algorithm, segment routing and JumpFlow leverage source routing to reuse existing forwarding paths; thus, it reduces the number of existing flows in a network. STAR considers switch memory utilization as a cost metric for forwarding path calculation, and it avoids switch memory overflow. LiteVisor instead aggregates the paths that have similar host locations (to be explained in Section III). Second, in terms of who does the path calculation, previous studies require controllers to do the calculation, whereas LiteVisor makes no change to controllers but receives the calculated paths from tenant controllers and aggregates them in a network hypervisor. Third, segment routing, JumpFlow, and STAR assume non-virtualized SDN environments. Thus, it is not clear whether their techniques are applicable in virtualized environments.

Another forwarding method, TRILL [31], is a layer-2 based forwarding technique. Each TRILL switch operates using link state protocol that propagates all of the TRILL switch information among each other. In other words, all of the TRILL switches have the information necessary to reach the final TRILL switch. Thus, once any packet enters into a TRILL-based forwarding network, the first ingress switch encapsulates the packet with the last TRILL switch identifier.

TRILL is based on non-SDN networks which means that each switch decides the forwarding path in a distributed
way. SDN removes this control complexity by deciding all forwarding paths with a centralized controller. Thus, TRILL is not directly applicable to the SDN-based network virtualization.

Finally, the Tag-in-tag method [26] reduces the number of flow rules by attaching two tags to each packet header, which is similar to a tunneling technique. In the forwarding used by Tag-in-tag, each packet has a path tag that represents the forwarding flow of the packet; this path tag is matched by intermediate (core) switches and a flow tag that notifies the end-host to forward packets to the destination at the edge switches. Moreover, AggreFlow [32] introduced flow-set routing that reduces the number of flows by aggregating flows which have the same hash value of five tuples, so flows sharing the same hash values go through the same path and are aggregated. For both studies, the network switch should be modified to cover non-standard operations. Tag-in-tag requires a new type of tunnel header, and AggreFlow requires edge switches to forward packets based on hash operations (e.g., CRC32).

All of these studies solve the memory problem in their own ways, but they are not applicable to the SDN-based network virtualization directly. In the virtualized environment, the forwarding path of each tenant is restricted in its virtual network topology and should be isolated from other tenants. Thus, the unit of the aggregation should be restricted to the flows belonging to the same tenant to accommodate the aforementioned condition. Moreover, the paths are given from each tenant controller, so the calculation of forwarding paths is not the role of network hypervisors. In addition, several studies require multiple tunneling, or tunneling modification in the middle of packet forwarding, which can cause performance degradation. Finally, it is desirable not to modify existing SDN switches for the effective deployment of the solution.

C. NETWORK RECONFIGURATION FOR VM MIGRATION

Regarding VM migration, the method of network reconfiguration relies on the host VM monitor. Thus, in the case of KVM [33], the available range of migration is restricted to the same subnet of the source subnet. Hyper-V [34] requires the source and destination host to be on the same physical or virtual networks, which implies that they should either be on the same subnet or communicate via a tunnel. For both solutions, reverse address resolution protocol (ARP) is used to update the mapping between the IP address of the host and the new MAC address. This generates significant network traffic and long delays.

SDN-based solutions have also been proposed to address the above shortcomings. First, Pupatwibul et al. [35] implemented mobile IP with SDN, by additionally introducing a state_update message into the OF protocol. When a new host connects to a switch, the switch reports the updated event to the controller and the controller installs the reported event. In addition, Tantayakul et al. [36] implemented PMIPv6 based on SDN with the observation that the signaling process between a local mobility anchor and a mobile access gateway in the PMIPv6 is similar to the process of installing a flow rule between a switch and the controller. Wang et al. [37] introduced an address separation method that uses the IP address of the arbitrary switch, which belongs to the forwarding flow of each packet, as its identifier address. Then, the SDN controller rewrites the IP address at the InpSwitch and restore the address at the EpSwitch. VIRO [38] was proposed as a similar address separation method, but the allocated address is derived from the virtual binary tree, which is generated from the physical topology; therefore, a new protocol header was proposed to contain the allocated address. These studies have addressed the disadvantages of previous network reconfiguration techniques (e.g., significant network traffic and long delays) well.

For SDN-based network virtualization, [39] proposed the reconfigurable architecture of a network hypervisor which consists of multiple distributed hypervisors. However, the study dealt with control path migration. To this end, previous studies have not considered VM(host) migration for SDN-based network virtualization.

In SDN-based network virtualization, network reconfiguration can be performed in two different cases: 1) tenant controller-driven and 2) network hypervisor-driven. In 1), the tenant controller is responsible for the network reconfiguration, and the network hypervisor should intervene as follows. The tenant controller should receive the migration notification prior to its reconfiguration process. So the network hypervisor translates the information related to VM migration and delivers it to controllers, because when the cloud orchestration system notifies VM migration, the host address and future location of the host and switch to be migrated should be translated to the virtual address and virtual switches. This information is only available from the network hypervisor. In addition, the flow rules for network reconfiguration, created from tenant controllers, should be translated from the virtual to the physical network context through the network hypervisor.

Alternatively, in 2) tenant controllers are not involved in network reconfiguration, but the network hypervisor performs all the required tasks. An advantage of 2) is that it does not require any event delivery or the virtual-to-physical translation, which can further reduce the network reconfiguration delay and gain additional advantages such as flow rule aggregation. Therefore, architecturally, it makes sense for the network hypervisor to support network reconfiguration. Moreover, the network hypervisor enables seamless network reconfiguration by hiding the migration event, managing the movement of each host, and updating the new flow rules after the migration, giving the tenant more freedom to choose its controller regardless of whether or not it supports network reconfiguration. Thus, this paper designs network reconfiguration into LiteVisor.

D. MOTIVATION AND GOAL

To reduce the shortage of switch memory, previous studies proposed various flow aggregation techniques based on the
forwarding path calculation, multiple tunneling, or modifications of an existing switch’s operations. This basis makes the application of existing techniques challenging in the SDN-based network virtualization environment where the forwarding paths are calculated from tenants. In addition, because the cloud datacenter is performance-sensitive, the number of packet encapsulation and decapsulation should be minimized. However, because already datacenters have its SDN physical network, it should be compatible to the standard SDN switch specifications, easing the difficulty of their deployment.

In addition, VM migration is an essential and demand-increasing technology in datacenters for edge computing [40], which involves network reconfiguration. However, network reconfiguration for SDN-based network virtualization has not been studied. Moreover, both flow aggregation and flow reconfiguration for VM migration have never been studied together. Hence, this paper sets the goals as follows.

- To provide flow aggregation in the context of the virtualized SDN, flow aggregation should occur based on forwarding paths given from a tenant. In addition, flow after aggregation should function within a virtual network topology. To satisfy this condition, LiteVisor works between flows belonging to the same tenant.
- To support efficient VM migration in SDN-based network virtualization, we aim to provide seamless network reconfiguration.
- Our study aims to propose an efficient forwarding semantic to provide flow aggregation and network reconfiguration simultaneously.

III. DESIGN

In this section, we explain the design of LiteVisor. We first present the overall architecture of LiteVisor. Then, the concept of a forwarding scheme for LiteVisor to enable flow aggregation and seamless network reconfiguration is proposed. In addition, we illustrate the detailed mechanisms for flow aggregation and network reconfiguration used by LiteVisor. Moreover, we discuss how to solve issues from flow aggregation and network reconfiguration.

A. LITEVISOR ARCHITECTURE

Figure 3 depicts the overall architecture of LiteVisor. When a tenant controller sends a flow rule, LiteVisor translates the vSDN flow information (vFlowInfo) from the message to pSDN flow information (pFlowInfo). The pFlowInfo and vFlowInfo consist of source and destination hosts, InpSwitch and EpSwitch which identify location information, and InARPSwitches. The conversion between virtual and physical flow is carried out in the virtual flow builder, followed by the physical flow builder in Figure 3. LiteVisor achieves the flow aggregation and seamless network reconfiguration with the Locator, Identifier, and Tenant sEparating (LITE) scheme, which will be discussed in Section III-B. The LITE Manager (LManager) translates the pFlowInfo into LITE-based flow rules. During the process, LManager aggregates multiple flows traversing the same path into one flow. In addition, when the cloud orchestration system notifies a migration event after the installation of LITE-based forwarding rules, LiteVisor migration manager recalculates the forwarding flow for the new location of a VM. This hides the host migration from the tenant controller by calculating a new optimal physical flow for the host.

B. LITE AND LITE-BASED FORWARDING

1) LITE

The addressing scheme in an IP network architecture does not distinguish between the identifier and locator of each host [41]. In other words, the IP address is used in both cases: 1) to distinguish entities at the edge switches to finally forward packets, and 2) to find the location of the host for packet routing. However, this integrated role of IP address leads to the following problems [42], [43]. When the host migrates to another point on the network, the IP address should be altered because the location is changed. Even if the network architecture permits an arbitrary IP address without consideration of the host location, all the switches in the network would need to update the routing table to the newer IP address to forward the packets.

Locator-Identifier Separation Protocol (LISP) [44] resolves the problem by applying two kinds of IP addresses, such as the endpoint identifier (EID) and the routing locator (RLOC). EID is used for identifying a host, while RLOC is for routing the packets over the Internet. The end host functions as it currently does, and the assigned IP address of the host becomes an EID. The EID can be allocated independently of the network topology and is used to specify the host. Conversely, the RLOC is topologically assigned and used for packet routing and forwarding. The actual packet forwarding is performed using the RLOC addresses. Although LISP protocol specification defines its tunnel header, RLOC address can be contained in any tunneling protocol header.

However, in network virtualization, LISP allows all connections belonging to the same RLOC pair (InpSwitch and EpSwitch) pass through the same path without isolation between the tenants. This means that the forwarding path of the packet, which is the result of each tenant’s own routing, is not isolated between tenants. In order to utilize the advantages of LISP and the isolation between virtual networks simultaneously, it is necessary to distinguish the RLOC for LISP and the TID for a virtual network when transmitting the packet. There are two approaches to achieve this. First, based on existing tunneling protocols such as VLAN and VxLAN, each packet can be tunneled with TID and RLOC respectively (double-tunneled) in tunnel header. However, this approach incurs significant overheads for packet encapsulation and decapsulation [45]. Moreover, because the double-tunneled packet should be matched twice in each switch to match TID and RLOC, respectively, additional performance degradation is unavoidable.


10.1109/ACCESS.2018.2877416, IEEE Access
Thus, we design an alternative approach, which integrates network virtualization and LISP, called Locator, Identifier, and Tenant sEparating (LITE). LITE uses a new form of an identifier, called isolated RLOC (iRLOC), which integrates TID and RLOC together: iRLOC can be defined and contained in each packet with various tunneling protocol. If iRLOC is implemented with IP header, iRLOC is in IP address form - the RLOC of the pSwitch in the upper 24 bits and TID in the lower 8 bits. Note that lower 8 bits in IP address specify a host, but pSwitch does not need it. For example, if the RLOC is 1 and the TID is 1, the iRLOC would be 0.0.1.1. Moreover, if the LiteVisor implements iRLOC with VXLAN [7] which has 24-bits ID field, the upper 16 bits can be used for RLOC and lower 8 bits for the TID. LiteVisor can make iRLOC flexibly according to the address space size of TID and RLOC in the given network.

2) LITE-based Packet Forwarding

LITE-based packet forwarding (LForwarding) is conducted by adding and removing the tunnel header, which includes the iRLOCs of InpSwitch and EpSwitch. Note that iRLOC allows LForwarding to be implemented by any arbitrary tunneling protocols (e.g., VXLAN [7], GRE [46], MPLS [47], LISP [44]). Throughout this paper, we use the IP address format to express iRLOC for illustrating designs of LiteVisor.

Figure 4 depicts an example topology of the pSDN mapped by the vSDN1 (TID1). The vMAC and the vIP of Host1 are 00:00:00:00:00:01 and 10.0.0.1, respectively, and those of Host2 are 00:00:00:00:00:02 and 10.0.0.2, respectively. It is assumed that the iRLOCs of each pSwitch are from 0.0.1.1 and 0.0.4.1. The flow table and packet format of InpSwitch, InterpSwitch, and EpSwitch are shown in Figure 5, Figure 6, and Figure 7 respectively.

When Host1 sends packets to Host2, Host1 is the source and pSwitch1 is the InpSwitch connected to Host1. Host2 is the destination, and pSwitch4 is the EpSwitch. Thus, InpSwitch performs packet encapsulation for LForwarding. In Figure 5(a), SclIPv4 in Action is the iRLOC of pSwitch1 and DstIPv4 in Action is the iRLOC of pSwitch4. After the packets sent by Host1 pass through pSwitch1, they are encapsulated with the tunnel header as illustrated in Figure 5(b). In the case of pSwitch2 and pSwitch3 (Figure 6), the packets that are transmitted to the next switch only refer to
the iROCs in the SrcIPv4 and DstIPv4 address fields with the match field of flow rules for pSwitch2 and pSwitch3 configured as follows: \( \text{SRC}_{\text{iROC}} = 0.0.1.1, \text{DST}_{\text{iROC}} = 0.0.4.1 \). When the encapsulated packets reach pSwitch4, this switch decapsulates the packets via decapsulation action and sends them to Host2 on the basis of the flow table. The entry must include the vMACs of Host1 and Host2 to classify the output port connected to Host2 after decapsulation, as shown in Figure 7(a). Then, as shown in Figure 7(b), the decapsulated packets reach Host2.

C. FLOW AGGREGATION

Figure 8 illustrates the sequence of flow processing in LiteVisor. Each tenant controller creates flow rules for the packet forwarding within their virtual network. The flow aggregation of LiteVisor aims to merge flows sharing the same edge switches into single flow between edge switches. To achieve this aggregation, LManager in Figure 3 manages the location information of hosts. Whenever a flow rule arrives from a tenant controller (③), virtual flow builder collects the flow rules and creates a vFlowInfo (②). Then, physical flow builder translates the vFlowInfo into a pFlowInfo by translating virtual addresses into physical addresses in regard to topology and address virtualization (④). Then, LManager finds the location of the end-host pair of the pFlowInfo (⑤), which are the iROCs of the InpSwitch and EpSwitch. Throughout this paper, the location per flow is denoted as an iRLOCpair \( (iRLOC_{\text{InpSwitch}}, iRLOC_{\text{EpSwitch}}) \).

The aggregation is achieved with the iRLOCpair as follows. LiteVisor checks whether the iRLOCpair already existed before (⑥). If the pair has not been created before, it means that the first forwarding path between an iRLOCpair needs to be created. Thus, LManager installs all the required flow rules for a given path. The flow rules installed between an iRLOCpair, which are installed at InterpSwitches, match only the tunnel header’s iROCs and forwards packets (⑦). For InpSwitch, LManager installs a flow rule that encapsulates the packet with the iROCs at the tunnel header. In addition, a flow rule for decapsulation is installed at the EpSwitch (⑧). Finally, LManager creates a mapping between the pFlowInfo and iRLOCpair (⑨) and marks the pFlowInfo that causes to create the iRLOCpair for the first time as the original pFlowInfo \( (OpFlowInfo) \) to deal with network reconfiguration efficiently, which will be explained in Section III-D.

On the other hand, when the created iRLOCpair already exists (⑤), LManager omits the flow rule installations for the InterpSwitches. Thus, the flow rules for the current pFlowInfo in the InterpSwitches are merged into the flow rules installed previously. LManager installs flow rules only for the InpSwitch and EpSwitch that encapsulate and decapsulate the packet with the tunnel header of iROCs (⑧). In this manner, a number of flows between the same iRLOCpair can be aggregated by a single flow; therefore, the memory consumption of InterpSwitches is reduced, and Section IV presents the effectiveness of LiteVisor.

D. SEAMLESS NETWORK RECONFIGURATION

LiteVisor supports seamless network reconfiguration for VM migration. Note that this paper focuses on network reconfiguration after migration. We assume that VM migration is performed with existing techniques. To that end, we design the network reconfiguration of LiteVisor to be compatible with existing migration techniques. Since notification of VM migration is generated by the cloud orchestration system [48], the migration manager of LiteVisor is responsible for providing support for the network reconfiguration. If a host is connected to a new switch, the migration manager calculates all possible pFlowInfos for ingress and ingress traffic related to the new switch and selects the new optimal pFlowInfo \( \text{(PFlowInfo}_{\text{new}}) \), which is explained in detail below.

When a VM migrates, the edge switch to which the VM is connected is changed. Therefore, the migration manager in Figure 3 determines whether to allocate a new iRLOCpair \( (iRLOCpair_{\text{new}}) \) or refer to an existing iRLOCpair. In addition, removing an old iRLOCpair \( (iRLOCpair_{\text{old}}) \) requires the consideration of other pFlowInfos that are already mapped to it \( \text{(PFlowInfo}_{\text{old}}) \). The migration manager achieves this by using Alg. 1.

Suppose that \( iRLOCpair_{\text{new}} \) for \( \text{PFlowInfo}_{\text{new}} \) differs from \( iRLOCpair_{\text{old}} \) for \( \text{PFlowInfo}_{\text{old}} \). The procedure stated in Alg. 1 generates an \( iRLOCpair_{\text{new}} \) and updates related objects (e.g., iRLOCpair and pFlowInfo).
Input: pFlowInfo_{new}, pFlowInfo_{old}, 
    iRLOCpairStore 

Output: iRLOCpair_{new}, pFlowInfo_{old} 

iRLOCpair_{old} = makeiRLOCpair(pFlowInfo_{old})  
iRLOCpair_{new} = makeiRLOCpair(pFlowInfo_{new}) 

if pFlowInfo_{old} is OpFlowInfo of iRLOCpair_{old} then 
    // Case 1 
    if other pFlowInfos do not reference iRLOCpair_{old} then 
        // Case 2 
        remove pFlowInfo_{old} from iRLOCpairStore 
    else 
        // Routine 1 
        store iRLOCpair_{new} to iRLOCpairStore 
        set pFlowInfo_{new} to OpFlowInfo of iRLOCpair_{new} 
    end 
    else 
        // Routine 2 
        pFlowInfo_{new} references to iRLOCpair_{new} 
    end 
else 
    change OpFlowInfo of iRLOCpair_{old} 
    update other pFlowInfos referencing to pFlowInfo_{old} 
    if iRLOCpair_{new} is not in iRLOCpairStore then 
        // Routine 1 
    else 
        // Routine 2 
    end 
end 
else 
    remove reference pFlowInfo_{old} from iRLOCpair_{old} 
    if iRLOCpair_{new} is not in iRLOCpairStore then 
        // Routine 1 
    else 
        // Routine 2 
    end 
end 

Algorithm 1. Algorithm for allocation of iRLOCpair after migration

At first, Alg. 1 checks whether pFlowInfo_{old} is the OpFlowInfo of the iRLOCpair_{old} (Case 1). If Case 1 is true, migration manager checks whether other pFlowInfo are referencing iRLOCpair_{old} (Case 2). If none of the pFlowInfo reference iRLOCpair_{old}, migration manager removes iRLOCpair_{old} from its own iRLOCpairStore; then, if iRLOCpair_{new} is not in iRLOCpairStore, migration manager stores iRLOCpair_{new} to iRLOCpairStore and pFlowInfo_{new} becomes the OpFlowInfo of iRLOCpair_{new} (Routine 1). If iRLOCpair_{new} is in iRLOCpairStore, pFlowInfo_{new} references to iRLOCpair_{new} (Routine2). If Case 2 is true, migration manager reselects the OpFlowInfo of iRLOCpair_{old} among the pFlowInfo referencing iRLOCpair_{old}, and updates the pFlowInfo except for the selected pFlowInfo because the OpFlowInfo is changed. Routine 1 and Routine 2 are performed for iRLOCpair_{new} If Case 1 is false, migration manager removes the reference to pFlowInfo_{old} from iRLOCpair_{old}. Routine 1 and Routine 2 are also performed for iRLOCpair_{new}.

Then, LiteVisor installs the flow table of pSwitches for pFlowInfo_{new} before migration is over. If there are some pFlowInfos referencing the iRLOCpair_{old}, the flow entries of pSwitches for pFlowInfo_{old} will not be removed; otherwise, they will be removed. In addition, if the OpFlowInfo of iRLOCpair_{new} exists, packets traversing pFlowInfo_{new} can be forwarded without additional flow rule installations because the flow table of pSwitches in pFlowInfo_{new} is already installed. If not, migration manager will establish the flow rule entries for iRLOCpair_{new}.

Regarding the complexity of this algorithm, we first check whether the pFlowInfo_{old} is the first forwarding path of iRLOCpair_{old} by a boolean variable that indicates whether the pFlowInfo caused the iRLOCpair to be created for the first time. If the pFlowInfo_{old} is not the first pFlowInfo, finding the other existing pFlowInfo is not required, so the configuration is done at O(1). If the pFlowInfo_{old} is the first forwarding path of iRLOCpair_{old}, the complexity of this algorithm can be O(n) when the number of existing pFlowInfos is n. To avoid this complexity, we add a counter for each iRLOCpair that counts the number of referencing pFlowInfos, so the complexity of the algorithm becomes O(1) as well.

E. CONSIDERATIONS
Changing the packet forwarding path for network reconfiguration brings up new considerations for network management. We design techniques to process an ARP request sent by a migrated host and simple packet loss prevention for stateless protocols such as UDP.

1) Proxy ARP support
Hosts periodically broadcast ARP requests. In SDN, most controllers generally establish a flow rule of pSwitches to enable ARP packets to be forwarded to the controller. Then, the controller performs proxy ARP, which is supported by several controllers such as ONOS. However, when a host is migrated, a controller cannot notify the event because LiteVisor hides the physical flow modification from a tenant controller in order to provide seamless network virtualization; thus, the ARP request cannot be appropriately processed. To overcome this problem, LiteVisor supports proxy ARP instead of the tenant controller. When LiteVisor receives an ARP request, it checks whether the request is related to the migrated host; if yes, it is not forwarded to the tenant controller. Instead, LiteVisor builds an ARP reply and sends...
a direction for delivering the ARP request to the destination at the pSwitch that sent the request message.

2) Packet loss prevention

During network reconfiguration, packet loss may happen. Hence, LiteVisor employs a simple packet buffering technique. Note that LiteVisor covers loss prevention only during the network reconfiguration as LiteVisor is responsible for the network reconfiguration. Note that this packet loss prevention is required only for stateless transport protocols which do not perform packet loss management themselves (e.g., UDP).

When the migration is initiated, packets of all flows related to the host are forwarded to LiteVisor as migration manager checks all pFlowInfos related to the destination host and deletes the flow table entry of the InpSwitch of pFlowInfos affected by migration (the affected pFlowInfos are identified by Alg. 1). Then, the InpSwitch forwards flow rule request messages containing packets destined for the host to LiteVisor because the switch cannot decide which actions to apply for the packet. When LiteVisor receives the flow rule request messages with unprocessed packets, LiteVisor stores the packets from the flow rule request messages in the buffer of pFlowInfo related to the migrated host. After complete migration, LiteVisor forwards the buffered packets to the EpSwitch of pFlowInformew.

Figure 9 provides the overview of the packet loss prevention of LiteVisor. We focus on only the ingress traffic to the host because there is no egress traffic during migration. Assume that Host5 sends packets to Host6 and they are connected to pSwitch1 through pSwitch4. At this moment, the iRLOCpair is (pSwitch1, pSwitch4), and pFlowInfo1 consists of Host5, pSwitch1, pSwitch2, pSwitch3, and pSwitch4. Furthermore, it is assumed that pFlowInfo1 is the OpFlowInfo of iRLOCpair (pSwitch1, pSwitch4).

If Host6 migrates from pSwitch4 to pSwitch3, the cloud orchestration system in Figure 3 will initiate migration. The migration manager searches pFlowInfos related to Host6. If pFlowInfo1 is found, in order to remove the entry of pSwitch related to pFlowInfo1, the migration manager sends a flow rule deletion command to pSwitch1, which is the ImpSwitch of pFlowInfo1. Subsequently, pSwitch1 forwards a flow rule request message including the packet destined to Host6 whenever it receives packets to notify the unprocessed packet on the switch. When migration manager receives them, the migration manager stores the packets to the buffer of pFlowInfo1. After migration, the migration manager directly sends a control message to deliver buffered packets to the destination at pSwitch3, which is an EpSwitch of pFlowInformew. The destination host does not need to be aware of packet loss technique, because the buffered packet comes in from the EpSwitch switch, just like any other packets. Now, pFlowInformew is pFlowInformo2, and it consists of pSwitch1, pSwitch2, and pSwitch3. It is denoted as pFlowInfo1 (Host5, pSwitch1, pSwitch2, pSwitch3, Host6) and the iRLOCpairnew is (0.0.1.1, 0.0.4.1).

Other considerations for designing packet buffering are the size of the buffer and rate limiting to resolve the situation where many flows send packets at high bitrates. This is a well-known research issue in networking, and significant research attention has been given to queuing theory. We do not extend the buffering technique because the focus of our work is the reconfiguration mechanism; however, the proposed mechanism can be implemented with the existing rate limiting techniques [49].

IV. EVALUATION

In this section, we explain the implementation of LiteVisor and the experiments to validate our design. LiteVisor implements the whole components of Figure 3 on OpenVirteX. We develop LiteVisor based on OF version 1.3, a widely used southbound interface. To implement LForwarding scheme in LManager, we choose MPLS protocol as the tunnel header because MPLS is supported from OF version 1.1. However, the designs in this paper can be adapted to any other southbound interfaces and tunneling protocols. For example, when this scheme is implemented with OF version 1.0,
LForwarding can be realized with the VLAN protocol that OF version 1.0 supports, and with P4, new tunneling syntax and mechanisms can be developed for LForwarding.

With our implementation, we evaluate the effectiveness of the flow aggregation and seamless flow reconfiguration of LiteVisor. We set three physical servers connected to a 10 Gbps switch. Each server runs Mininet [50] for emulating physical networks, and LiteVisor and ONOS for tenant controllers. We test four network topologies: one linear topology, and three fat-tree topologies. Linear topology consists of several physical switches and hosts (Figure 10(a)). For fat-tree topology (Figure 10(b)), we use 4-ary (4 core, 4 aggregation, 4 edge switches, and 12 hosts), 6-ary (9 core, 18 aggregation, 18 edge switches, and 54 hosts) and 8-ary (16 core, 32 aggregation, 32 edge switches, and 128 hosts) topologies.

### A. FLOW AGGREGATION

LiteVisor enables flow aggregation based on LForwarding. We use four topologies as discussed in Section IV. We evaluate the average number of flow table entries of all pSwitches installed by OVX and LiteVisor under the condition that hosts send UDP packets with each other between its paired host. Among various network hypervisors, OVX and ONvisor have been proposed as open-source network hypervisors. We analyzed the implementation of these hypervisors, and found that they translate and install a flow rule from the virtual network controller in one-to-one manner. Therefore, the number of flow rules to be installed at physical switches are identical.

First, for linear topology, we set the number of pairs of fan-out hosts to 2, 6, and 10. In addition, we vary the number of InterpSwitches from 1 to 10. Figure 11 shows the number of flow entries between OVX and LiteVisor, and Figure 12 summarizes the reduction rate varying the number of fan-out hosts and InterpSwitches. In case of OVX, it establishes two entries per pair of hosts for ingress and egress traffic. On the other hand, LiteVisor installs two entries per pair of hosts in the InpSwitch and EpSwitch; however, the pairs of hosts that belong to the same iRLOCpair need only two entries in InterpSwitches regardless of the number of fan-out hosts. Thus, the reduction rate in the number of entries depends on the number of InterpSwitches. Compared to OVX, the number of LiteVisor entries is reduced to 1/4 (75% reduction) when the number of pairs of hosts is 10, and the number of InterpSwitches is 10. The experiments show that the minimum reduction is about 17% when there are two fan-out hosts with one InterpSwitch.

Second, for the fat-tree topologies, we show the number of flow rules per switch layer in each topology (Figure 13). Because LiteVisor must still maintain two flow entries for packet encapsulation and decapsulation as OVX, the number of flow rules at the edge switch layer is identical. For core and aggregation switches, the reduction becomes higher when the size of the topology becomes bigger. The highest reduction rates at the core and edge switches are 75% and 88% respectively in 8-ary topology. In addition, 4-ary topology showed 50% and 75% reduction which are the minimum reduction rates in evaluations on fat-tree topologies. Hence, the results indicate that LiteVisor achieves significant reduction when many hosts have the same iRLOCpair.

### B. SEAMLESS NETWORK RECONFIGURATION

First, we show the network reconfiguration time \( T_r \) in linear topology varying the number of InterpSwitches and packet sending interval. Host2 migrates from pSwitchC to pSwitchB...
FIGURE 13. Comparing the average number of entries in fat-tree topologies

FIGURE 14. VM migration scenario

FIGURE 15. Reconfiguration time varying the number of InterpSwitches in linear topology (ms)

FIGURE 16. Reconfiguration time in fat-tree topologies (ms)

(Figure 14) while continuously sending UDP packets to Host1 with a regular sending interval.

To see the effects of the sending interval and number of switches, we change the value to 10, 30, and 50 ms. In [51], the inter-arrival times of flows are between 0 and 100 ms, so we choose three values within that range. In addition, to discuss the effect of the topology size on $T_r$, we increase the number of InterpSwitches to 3, 6, 9 and 12, as the overall length of the topology increases with the number of InterpSwitches, and test against each selected sending interval.

We run each experiment ten times and calculate the standard errors. In addition, only LiteVisor is measured because OVX cannot support network reconfiguration at all. Figure IV-A shows the average measurement and standard errors of $T_r$ as a function of the number of InterpSwitches and the sending interval. It shows that $T_r$ increases as the number of InterpSwitches increases. Changing the sending interval has a relatively little effect on $T_r$.

For fat-tree topologies, we measure the average $T_r$ and standard error values where the VM migration occurs for both intra-pod and inter-pod (Figure IV-A). We alter the UDP packet sending interval to 10, 30, and 50 ms, and run each evaluation ten times, as the linear topology. Similar to the linear topology, when the scale of the topology becomes bigger, $T_r$ also increases. The reason is that $T_r$ contains the time for recalculating path between migrated hosts using the shortest path algorithm (e.g., Dijkstra algorithm) and installing new flow rules. Thus, the $T_r$ increases as the...
Although the migration time, excepting the network reconfiguration, is 2 s, and the migration is initiated at 5 s. When the host is migrated to a new switch, and the host sends data packets, the new switch sends flow rule requests including data packets. Then, LiteVisor stores the packets included in the flow rule request messages to its buffer of pFlowInfo. After migration including network reconfiguration is complete, LiteVisor forwards and delivers buffered packets to the switch to which Host is migrated. Figure 17 shows that all UDP packets reach their destination without any packet loss, which is the result of the packet buffering. As the LiteVisor sends the buffered packet to the migrated switch within approximately 2 s after the notification of a migration event, the UDP sequence increases vertically at the end of migration. In terms of throughput, Figure 18 shows that the throughput increases sharply after migration due to the transmission of buffered packets.

V. DISCUSSION
A. ISOLATION OF AN AGGREGATED FLOW
With the LITE-based forwarding, the flows are aggregated only when the source and destination host locations are identical, and the flows belonging to the same tenant. If some tenants want separate control on intermediate hops after it is aggregated, because the intermediate hops are bypassed, the only way is disaggregation. Currently, LiteVisor does not support disaggregation. However, LiteVisor can be extended to go back to flows before aggregation as follows. LiteVisor keeps all the flow rules and path information from tenant controllers after aggregation. When a tenant needs disaggregation, LiteVisor can retrieve the corresponding flow rules for the request and install the flow rules separately.

B. TUNNELING OVERHEADS
For network virtualization, packet encapsulation and decapsulation, so known as tunneling, is essential to provide isolation between various tenants. For example, FlowN [4], which is an SDN-based network hypervisor, leverages VLAN. VFP (Microsoft) [52] and NVP (VMware) [53], which are widely used industrial network virtualization solutions, also deploy single tunneling for every packet. In other words, at least single tunneling is inevitable for providing network virtualization efficiently. LiteVisor leverages the tunneling operation for providing flow aggregation and network reconfiguration in addition to network virtualization. Thus, LiteVisor does not add additional tunneling but requires a single tunneling per packet.

Compared to network hypervisors that do not use tunneling for virtualization, LiteVisor uses tunneling that incurs en/decapsulation delay and packet fragmentation. However, these overheads can be reduced with the offloading of tunneling protocols on the NIC (e.g., VLAN or STT [53]).

In addition, because the LITE forwarding requires only the outer header created by the network hypervisor, rather than modifying or using the original packet header information, tenants can still use tunneling for their own purposes. Further, to isolate the packet forwarding between tunneled traffic, LiteVisor can allocate different iRLOCs for each tunneled flow that has the same edge switch pairs but with a different tunnel header. For example, when the tenant deploys VLAN to separate their traffic, LiteVisor encapsulates and decapsulates the VLAN-tunneled packet. In other words, when LiteVisor takes VLAN as its tunneling technique, LiteVisor performs VLAN encapsulation again for aggregation, which allows the different iRLOC identifiers to be allocated per (VLAN ID, ingress switch, egress switch) pair, and contained at the outer-most LiteVisor tunnel header.

C. SECURITY
Let us discuss the security aspects of LiteVisor. LiteVisor aggregates flows within a single tenant, and the flow information is isolated between tenants. Thus, the information is not shared between tenants. However, if a malicious tenant provides incorrect host addresses, there is a chance that LiteVisor will aggregate incorrectly, leading to packets not being delivered properly. However, this security problem emerges from the tenant itself, not from the techniques of LiteVisor. In other words, aggregation itself does not compromise the contents of the packets. Therefore, LiteVisor does not create additional vulnerabilities.
VI. CONCLUSION
Various network hypervisors have been proposed to achieve SDN-based network virtualization. However, their forwarding techniques excessively consume the limited memory of pSwitches, and none of them supports network reconfiguration after VM migration. In this paper, we propose LiteVisor to support flow aggregation and seamless network reconfiguration after VM migration, which is achieved by introducing LITE-based forwarding while assuring isolation between vSDNs. We implement our design in an open-source network hypervisor, OVX, and evaluate the critical parameters (i.e., the number of flow table entries and reconfiguration time) against the performance of OVX. In our future work, we will study network optimization using the flow management design of LiteVisor.

REFERENCES


GYEONGSIK YANG received the B.S. and M.S. degrees in computer science from Korea University, Seoul, Korea in 2015 and 2017, respectively. He is currently a Ph.D. student at Korea University, Seoul, Korea.

He worked as a research intern at Microsoft Research Asia in 2018. His current research interests include network virtualization, datacenter networking, software-defined networking, and traffic engineering.

BONG-YEOL YU received the B.S. degree in computer science from Korea University, Seoul, Korea in 2016. He is currently an M.S. student at Korea University, Seoul, Korea.

His current research interests include network virtualization, programmable networks, datacenter networking, and software-defined networking.

SEONG-MUN KIM received BS degree in Computer Science from Dankook University in 2007, and M.S., and PhD degrees in Computer Science and Engineering from Korea University, Seoul, Korea in 2012 and 2015, respectively. He is currently a researcher at KT R&D center, Seoul, Korea.

His interests in research include distributed mobility management, mobility protocol design, software defined networking and performance analysis. Especially, Network slicing, Software defined infrastructure, SDN-based network virtualization and network hypervisor.

CHUCK YOO received the BS and MS degrees in electronic engineering from Seoul National University, and the MS and PhD degrees in computer science from the University of Michigan, Ann Arbor. He is now a professor in the College of Informatics at Korea University, Seoul, Korea.

He worked as a researcher in the Sun Microsystems Laboratory from 1990 to 1995. He has been with Korea University since 1995. His research interests include server/network virtualization and operating systems.