SVirt: A Substrate-agnostic SDN Virtualization Architecture for Multi-tenant Cloud

Yirong Yu, Dan Li, Yukai Huang

Tsinghua University

***Abstract*—Data center operators are accepting software defined networking (SDN) to manage their networks, but it remains challenging how to provide desirable virtual SDN services to tenants in a public cloud. We design SVirt, which enables highly flexible virtual SDN in a multi-tenant cloud by a substrate-agnostic SDN virtualization architecture. By redesigning the physical switch's processing pipeline with a "*late-binding key extractor*", SVirt supports virtual SDN switches with different processing pipelines simultaneously on a physical switch. In the control plane, SVirt enables "*many-to-one*" and "*one-to-many*" mapping when allocating the physical resource for a virtual network, which embraces arbitrary topology and TCAM resource demanded by a virtual network. In the data plane, SVirt explicitly carries the forwarding context information in the packets, overcoming the "*context-loss problem*" in a virtual SDN network. We develop a NetFPGA prototype of SVirt switch. Evaluations based on event-driven simulations and prototype-based experiments demonstrate that, compared with traditional approaches, SVirt significantly enhances the cloud's capability to accept various virtual SDN requests and improves the network's throughput.**

I. Introduction

Data center operators are accepting software defined networking (SDN) to manage their networks. A recent survey by Infonetics shows that 87% of medium and large North American enterprises intend to have SDN live in the data center by 2016 [1]. However, for many enterprises which prefer using public cloud services rather than building their own physical infrastructures, it remains challenging how to get desirable virtual SDN service from a multi-tenant cloud. An important requirement of SDN virtualization is to decouple a virtual network specification from the details of the physical network. Previous SDN virtualization solutions [5], [6], [7], [8] seek to achieve flexible topology and isolated flow space for a virtual network, but lack of consideration of the constraint to a virtual network caused by the processing logic and resources of individual physical switches.

Particularly, the physical network in today's cloud poses the following limits to a virtual network specification. First, the *packet processing pipeline* of a switch is fixed, either by hard-coded switching chips or by reconfigurable switching chips [3], [16]. But the processing logic required by tenants' virtual SDN switches can be various. It is difficult for a physical switch to simultaneously support virtual SDN switches with different processing pipelines. Second, when the number of virtual SDN switches required by a tenant is more than that of available physical switches, it is inevitable to map multiple virtual SDN switches from a virtual network into the same physical switch. In this kind of "*many-to-one*" mapping, it is insufficient to demultiplex packets by a virtual network identifier such as VLAN tag only. Third, if the demanded resource, *e.g.*, TCAM space, of a virtual SDN switch exceeds that of a physical switch, we need "*one-to-many*" mapping. Mapping a virtual SDN switch into multiple physical switches may cause the loss of in-pipeline information [2], *e.g.*, the meta_data exchanged between pipeline stages, the action set that is accumulated to execute at the end of the pipeline, *etc.*

In this work we design SVirt, a substrate-agnostic SDN virtualization architecture which addresses the problems above. SVirt abstracts the physical network as a single and large *network resource pool*. By enabling highly flexible mapping from a virtual SDN network to the physical network, SVirt masks the details of the physical network, such as the network topology, the number of switches, the packet processing pipeline and TCAM space in an individual switch, *etc.*, from tenants. A tenant gets his virtual SDN service based on whatever network topology and routing/security policy he defines on a virtual SDN switch; and runs his own SDN controller as well as manipulates the virtual SDN network as if occupying the whole network.

SVirt redesigns the processing pipeline of a physical switch to allow each stage of the pipeline to simultaneously match arbitrary packet header fields. A pipeline stage does not fix the matching fields of packet headers. Instead, a "*late-binding key extractor*" is introduced to determine the matching key of a packet by a matching field bitmap associated with every virtual SDN switch that resides in the pipeline stage. Consequently, virtual SDN switches with different matching fields can share the same pipeline stage. To obtain a packet's matching key with low latency and low cost, the late-binding key extractor divides the packet header fields into multiple 16-bit slots, and uses a selection network to extract the required packet header fields as a compact matching key. The process of the late-binding key extract takes only two time cycles.

In the control plane, SVirt enables many-to-one and one-to-many mapping when allocating virtual networks. A virtual network request will be rejected only when the physical network's overall resource cannot meet the requirement of the virtual network, not constrained by the physical network

The work was supported by the National Key Basic Research Program of China (973 program) under Grant 2014CB347800, the National Natural Science Foundation of China under Grant No.61170291, No.61432002, the National High-tech R&D Program of China (863 program) under Grant 2013AA013303, and Tsinghua University Initiative Scientific Research Program.

topology, the number of physical switches, or TCAM space in an individual physical switch. When allocating virtual networks, SVirt seeks to find the physical switch with the best-fitting TCAM width for a virtual SDN switch, so as to minimize the wasted TCAM space. When manipulating a virtual network (*e.g.*, installing forwarding entries into virtual SDN switches), SVirt controller plays as a translator between the tenant-specific SDN controller and the physical switches.

In the data plane, SVirt explicitly carries the forwarding context information of a virtual network in the packets, so as to address the "*context-loss problem*" which is particularly caused by many-to-one and one-to-many mapping. Every packet adds an SVirt header to pack the forwarding context information, such as the incoming virtual interface, the responsible virtual switch to handle the packet, the meta_data and action set exchanged between pipeline stages of a virtual switch residing in multiple physical switches, *etc.* This way, packets are correctly processed and forwarded within a virtual network, without the necessity to implicitly infer the forwarding context from the physical network. Besides, packets are forwarded in the physical network based on a separate forwarding fabric, and thus each virtual network enjoys the full header space [4].

We develop a NetFPGA prototype of SVirt switch. Event-driven simulations demonstrate that the late-binding key extractor in switches, the many-to-one and one-to-many mapping schemes, all significantly contribute to improving the cloud's capability to accept various virtual SDN requests and saving the physical switches' TCAM resource. Prototype-based experiments also show that, compared with traditional approaches, SVirt can greatly improve the network throughput and reduce the flow completion time in a busy cloud.

II. Problem and Previous Works

In this section we discuss the problems of substrate-agnostic SDN virtualization as well as previous solutions.

A. Model of Virtual Network

In a substrate-agnostic SDN virtualization architecture, a tenant issues a virtual network request with the virtual topology and all the virtual switches as follows.

- **Virtual topology**: $V = (N, L)$, where $N = \{S_1, S_2, ..., S_n\}$ is the set of virtual switches, and $L = \{E_1, E_2, ..., E_l\}$ is the set of virtual links. A tenant can define arbitrary topology for the virtual network, not constrained by the number of switches or the topology in the physical network.
- **Virtual switch**: $S = (F, M)$, where $F = f_1 f_2 ... f_k$ and $M = m_1 m_2 ... m_k$. f_i and m_i represent the matching packet header fields and memory space required by the i-th stage of the virtual switch's processing pipeline, respectively. A tenant can specify arbitrary matching fields and memory space for any virtual pipeline stage, not constrained by the number of pipeline stages, the processing logic, or the memory capacity in an individual physical switch. For simplicity, in this paper we primarily consider TCAM as the switches' memory.

We do not explicitly take performance-related requirements such as network throughput or latency of the virtual network into the model, since they are not the focus of this paper. But they will be taken into account in our virtual network allocation method (refer to Section III-B). We assume that all the physical switches support the same instructions and actions.

B. Previous Solutions

Previous works on SDN virtualization have many limitations in supporting substrate-agnostic virtual SDN. In FlowVisor [5], the topology of a virtual SDN network must be a subset of the physical network topology. OpenVirteX (OVX) [7] and VeRTIGO [8] go one step further by enabling more flexible virtual network topology. But they do not look inside switches. FlowN [6] not only supports flexible virtual topology, but also considers the maximum number of flow-table entries required by a virtual switch. However, FlowN does not discuss how to deal with the case when the number of flow-table entries required by a virtual switch exceeds the capacity of a physical switch. Both OVX and FlowN use a tenant identifier in the packet header to distinguish virtual networks in the data plane, which is insufficient to demultiplex the packets if we want to map two virtual switches from the same virtual network to one physical switch.

All the proposals above do not discuss the problem of how to support virtual switches with different processing pipelines in a physical switch. Existing reconfigurable switching chips [3], [10] cannot solve the problem, because the processing pipeline of a switching chip is still fixed at a time. POF [11] enables matching user-defined packet header fields in the switches, but it is not designed for SDN virtualization, and thus cannot meet the requirement of simultaneously supporting multiple different virtual pipelines on one physical switch. FlowAdapter [9] gives a possible solution by designing a hardware abstraction layer. It first converts the M stages of a virtual switch's pipeline into an equivalent one-stage flow table, and then converts the one-stage flow table again to suite the N stages of the physical switch's pipeline. However, the authors acknowledge that not all virtual pipelines can be converted in this way [9].

Many virtual network embedding algorithms [12], [13], [14], [15] are studied for mapping virtual networks to the physical network. For instance, M. Yu *et al.* [13] propose to split a virtual link into multiple virtual paths to achieve flexible virtual link mapping, while M. Chowdhury *et al.* [14] discuss virtual network embedding with coordinated node and link mapping. However, these works model the virtual network as a graph with bandwidth requirement on links and computation requirement on nodes. They do not consider issues such as multi-stage processing pipeline in switching nodes, and thus cannot be directly applied to SDN virtualization.

C. Problems to Solve

In order to map a substrate-agnostic virtual SDN network to the physical network, the new solution should not only be able to specify arbitrary virtual network topology and map

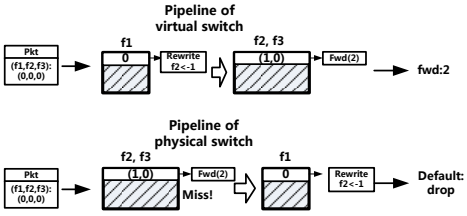


Fig. 1. Different orders of the pipeline stages cause different packet processing results.

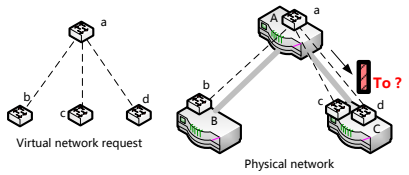


Fig. 2. Many-to-one mapping causes confusion in selecting the virtual switch to process an incoming packet on a physical switch.

multiple virtual switches to a physical switch, as previous SDN virtualization works do, but also support *flexible virtual pipelines* in one physical switch, mapping multiple virtual switches from the same virtual network to one physical switch (*many-to-one mapping*), as well as mapping one virtual switch to multiple physical switches (*one-to-many mapping*). Besides, a desirable SDN virtualization solution should provide full header space to each tenant.

Flexible virtual pipelines: We aim to support virtual switches with different pipeline logics simultaneously on one physical switch. There are two possible ways to achieve this. One approach is to use a single full matching table to encompass whatever processing pipelines required by a virtual switch. However, every entry needs to match all the possible packet header fields, and thus the Cartesian product explosion in transforming multiple stages of a virtual pipeline into a single table will make the memory space unaffordable. Moreover, a single matching table is not always feasible to describe the packet processing logic of a virtual switch [9].

The other approach is to use a multi-stage matching table, in which each stage only covers one or a few header fields, *e.g.*, destination IP address lookup, ACL five-tuple checking, *etc.* This method is also employed in recent commercial SDN switches [16]. However, the fixed sequence of the multi-stage pipeline in the physical switch will limit the sequence of the virtual pipeline on top of it, because it is not always possible to transform a multi-stage pipeline to another multi-stage pipeline with different order. As shown in Fig. 1, the virtual switch has a two-stage pipeline. The first stage matches field $f1$ (0) and modifies field $f2$ from 0 to 1, while the second stage matches field $f2$ (1) and $f3$ (0) and forwards to interface 2. However, if the two-stage pipeline in the physical switch has a different order, *i.e.*, matching $f2$ and $f3$ in the first stage and then matching $f1$ in the second stage, the packet will get dropped since there is no entry in the first stage to match the packet.

Many-to-one Mapping: When the number of virtual switches required by a virtual network is more than that of available physical switches, we need to map multiple virtual switches from the same virtual network to one physical switch. In this case, carrying a tenant identifier such as VLAN tag in the packet is not enough for the physical switch to locate the virtual switch that is responsible to handle the incoming packet. As shown in Fig. 2, two virtual switches c and d are mapped to the same physical switch C, both of which are connected to virtual switch a on physical switch A. When physical switch C receives an incoming packet from a, it cannot identify which virtual switch to process the packet, either by the in-packet tenant identifier or the incoming link (both virtual links of $a-c$ and $a-d$ are mapped to the same physical link).

One-to-many Mapping: When the TCAM space required by a virtual switch exceeds the capacity of a physical switch, or the number of pipeline stages required by a virtual switch is more than the available pipeline stages on a physical switch, we have to map one virtual switch into multiple physical switches. The idea of one-to-many mapping is not new and has been proposed in works such as [18], [19], [20]. However, in the scenario of SDN forwarding, forwarding context information is passed between different stages of a pipeline, such as the incoming interface of the packet, the meta_data exchanged between pipeline stages, the action set that is accumulated to execute at the end of the pipeline, *etc.* Loss of these information between physical switches will cause inconsistent forwarding context in different pipeline stages and accordingly incorrect result.

III. SVirt Design

We design SVirt, which provides substrate-agnostic virtual SDN service by solving the problems discussed in Section II.

A. SVirt Switch: Determining the Matching Fields in a Late-binding Way

Flexible Matching Table: To support flexible virtual pipelines on a physical switch, SVirt redesigns switches by introducing a new type of matching table, which we call *flexible matching table*, or *FMT*, as shown in Fig. 3. FMT leverages multi-stage processing pipeline to avoid the Cartesian product explosion problem in a single full matching table. Each FMT stage is equipped with a dedicated TCAM to avoid memory access contention between different stages. Unlike ordinary multi-stage matching table with hard-coded processing pipeline or reconfigurable pipeline but fixed at a time, a pipeline stage in FMT can match arbitrary packet header fields required by a virtual switch. FMT achieves this by a *late-binding key extractor*, which determines the matching fields of an incoming packet and extracts the matching key in a late-binding way.

Input and Output of the Late-binding Key Extractor: The core of FMT is the late-binding key extractor. To enable flexible matching fields in an FMT stage, we use a *matching field bitmap* to describe the packet header fields we want to extract for a certain virtual switch. For instance, a matching field bitmap of $00101(0)^x$ means that the third and fifth field of the packet header will be extracted as the matching key in an FMT stage for the corresponding virtual switch.

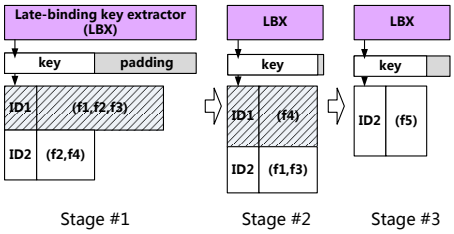


Fig. 3. Multi-stage flexible matching table (FMT) with late-binding key extractors.

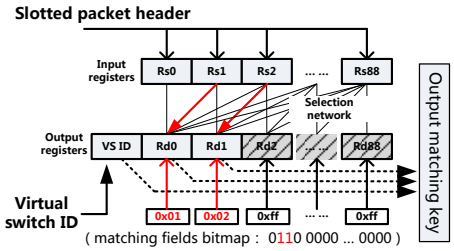


Fig. 4. Late-binding key extractor for an **FMT** stage.

In this way, SVirt not only supports well-defined protocol headers as those used in OpenFlow [2], but also supports user-defined protocol headers as in POF [11]. The input of the late-binding key extractor is the full header fields of the incoming packet, the virtual switch ID and the associated matching field bitmap; while the output is the matching key for the packet. Section III-B will discuss how to allocate the virtual switch ID and install the matching field bitmap by the SVirt controller. Section III-C will discuss how to obtain the virtual switch ID for an incoming packet.

An intuitive way to design the late-binding key extractor is to directly use the full packet header, the virtual switch ID and the matching field bitmap as the matching key. However, the full packet header may be much longer than the necessary packet header fields required and will thus cause significant waste of the TCAM space. For instance, in an IPv6/TCP header, the packet header length is more than 400 bits, while we may need only 48-bit destination MAC address in a stage. Therefore, we seek to use the least necessary packet header fields to form a compact matching key in the late-binding key extractor, and in what follows we describe how to realize it.

Static Templates or Dynamic Concatenation of the Matching Fields: There are two basic choices to obtain a compact matching key. First, we can enumerate all the possible combinations of the packet header fields as templates and statically put a key extractor for each template. When a packet comes, one specific extractor (indicated by the matching field bitmap) is selected to get the matching key. However, the number of such templates is exponential to the number of header fields, the cost of which is too high. Taking Openflow 1.4 as an example, where 41 packet header fields are supported, we need to put 2^{41} static key extractors. As a result, we turn to the second choice, which dynamically concatenates the packet header fields to extract the matching key. The memory required is only linear to the number of packet header fields.

Efficient Concatenation of the Packet Header Fields: There are several ways to complete the concatenation. A naive method is to get the required matching fields one by one from the packet header and make a fully compact key. But it results in high processing delay if the required key covers many header fields. Moreover, different virtual switches residing in an FMT stage may have different processing delays, which hurts the pipeline processing of the incoming packets.

A better way is to use a selection network to simultaneously extract the required fields and output to a series of registers. The selection network connects the packet header fields and a number of output registers as a complete bipartite graph. When processing the packets for a specific virtual switch, part of the multiplexers in the selection network are activated based on the matching field bitmap. Thus, the compact matching key is obtained with only two time cycles, one for getting the matching field bitmap and one for extracting the key from the selection network.

A consequent issue is how to set the length of an output register. The lengths of different packet header fields vary greatly, from 1-bit MPLS_BOS field to 128-bit source/destination IPv6 address field. Given that any packet header field can be output to any register, we can use the length of the widest packet header field, *e.g.*, 128-bit IPv6 source/destination address, as the length of all registers. But it causes considerable waste of the register space and TCAM space (in the flow table), since most packet header fields are much shorter than 128 bits.

To overcome this problem, we split every packet header field into multiple slots of equal length. A packet header field shorter than the slot occupies the whole slot with padding bits. The shorter each slot is, the compacter the output key is (with less padding bits), which also saves the register space and TCAM space. However, fine-grained splitting also causes quadratic growth of the cost of the selection network. By observing that most packet header fields are not longer than 16 bits and the longer ones are integral multiple of 16 bits, we choose 16 bits as the length of a slot. It can be regarded as a tradeoff between the cost of register/TCAM space and the cost of selection network.

Design of Late-binding Key Extractor: Fig. 4 shows our design of the late-binding key extractor. Each packet header field is divided into multiple input registers with 16 bits. The output registers are also of 16 bits. We still use OpenFlow 1.4 as an example. By inspecting the 41 packet header fields, we need 89 input registers. To encompass the worst case, we use an equal number of output registers. In the example shown in Fig. 4, the matching field bitmap of a virtual switch is $011(0)^{86}$, *i.e.*, the matching key requires the second and third slot of the packet header fields. Then the first two output registers, Rd_0 and Rd_1, will get activated to extract the second and third input registers, Rs_1 and Rs_2, respectively. In this way, we obtain the matching key as the virtual switch ID plus the first two output registers and padding bits (to match the pre-configured matching key length in the FMT stage).

It is worth noting that the width of the matching key in an FMT stage is reconfigurable when there is no active entry in the TCAM. For instance, a TCAM of 40-bit width with 32k

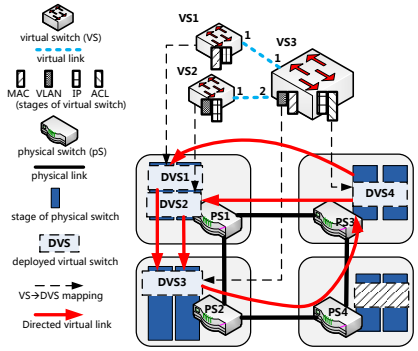


Fig. 5. Allocating the physical network resource for a virtual SDN network in SVirt.

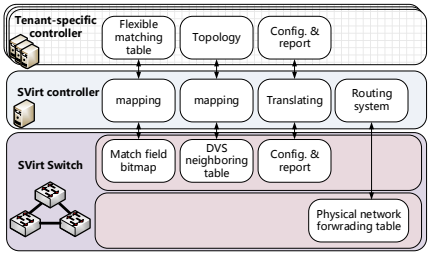


Fig. 6. Manipulating a virtual network in SVirt. The SVirt controller configures necessary states in the physical switches, and plays as a translator between the tenant-specific SDN controllers and the physical switches.

entries can be reconfigured as 80-bit width with 16k entries. But after allocating the TCAM space of an FMT stage for the first active virtual switch, the width is fixed until when all the active entries in the stage are cleared again.

B. SVirt Control Plane: Supporting Many-to-one and One-to-many Mapping

In the control plane, SVirt controller handles the virtual SDN request from a tenant and allocates the physical network resource. Besides, SVirt controller plays as a translator between the tenant-specific SDN controller and the physical switches when manipulating a virtual network. On one hand, it translates the virtual switch configuration requests from a tenant-specific SDN controller (*e.g.*, installing entries into the flow table of a virtual switch) to configuration request for physical switches; on the other hand, it translates physical switches' reports to tenant-specific SDN controllers (*e.g.*, the first packet of a flow).

Allocating Flexible Virtual Networks: When allocating the physical resource for a virtual network, SVirt supports many-to-one mapping and one-to-many mapping. A tenant's virtual network request will be rejected only when the physical network's overall resource cannot meet the demand of the virtual network, not limited by the physical network topology, or individual physical switch's pipeline logic or TCAM space.

We define *deployed virtual switch (DVS)* as an allocated instance in a physical switch for a full or part of a requested virtual switch. Since multiple virtual switches from the same virtual network can be mapped to one physical switch, a tenant identifier (such as VLAN tag) is not sufficient to differentiate the DVSes on a physical switch. Hence, the SVirt controller assigns a *globally unique ID* for every DVS allocated in the physical network. If a virtual switch is split into multiple DVSes in different physical switches, they also have different DVS IDs.

Fig. 5 shows an example of a virtual network allocation in SVirt. The physical network is composed of four physical switches, each with a two-stage available processing pipeline. A tenant requests a virtual network with three virtual switches, which are mapped to the corresponding physical switches as the figure illustrates. Both virtual switch *VS1* and *VS2* are mapped to physical switch *PS1*. Virtual switch *VS3* is split to reside in physical switch *PS2* and *PS3*, and thus there are two DVSes, namely, *DVS3* and *DVS4*, for it. In this kind of one-to-many mapping, a virtual link is divided into multiple directional ones due to the processing order. For instance, the virtual link from *VS3* to *VS1* is divided as *DVS3*→*DVS4* and *DVS4*→*DVS1*. A packet arriving at *VS3* is processed sequentially by *DVS3* and *DVS4*.

Given that virtual switches with different matching fields can share the same FMT stage of a physical switch, improper placement of virtual switches may lead to wasted TCAM space in physical switches. For example, an FMT stage has 200 entry space with 80-bit width. If we assign 100 entries for a virtual switch with 80-bit matching key and 100 entries for another virtual switch with 40-bit matching key, 25% of the TCAM space in the FMT stage will be wasted.

As a result, in SVirt when choosing the physical switch to host a virtual switch, we seek to minimize the wasted TCAM space in physical switches, with the constraint of performance metrics such as network throughput and latency. Since one-to-many mapping is supported, a chosen physical switch does not have to host the whole virtual switch; instead, it can host part of a virtual switch or even part of a virtual pipeline stage. We do not need to consider the specific type of matching fields when allocating a virtual pipeline stage, but do need to guarantee that the matching key width of the virtual pipeline stage is not greater than the TCAM width of the allocated FMT stage. From the physical switches that can host a virtual pipeline, we select the one that has the closest TCAM width to the virtual pipeline's TCAM width. The virtual network allocation fails only if none of the physical switches can host even a portion of the virtual switch due to TCAM width mismatch or TCAM space exhaustion.

Manipulating Virtual Networks: As shown in Fig. 6, in order to manipulate the virtual networks, the SVirt controller configures physical switches with necessary states for the virtual network. First, for every DVS of the virtual network, SVirt controller installs the matching field bitmap in each pipeline stage of the hosting physical switch, which is used by the late-binding key extractor. Second, for every DVS of the virtual network, SVirt controller installs a "*DVS neighboring table*" in the hosting physical switch, which records the hosting physical switch of each neighboring DVS in the virtual network, as well as the virtual interface ID towards the neighboring DVS. The DVS neighboring table is looked up when a physical switch sends out a packet in the physical network (refer to

Section III-C).

SVirt controller maintains all the mapping relationship between virtual networks and the physical network, which helps translating tenant's configuration request to physical switches and translating physical switch's report to tenant-specific SDN controller. A particular note is that the instruction of 'forwarding to interface i' configured by a tenant-specific SDN controller is translated as the instruction of 'forwarding to DVS x' installed in physical switches. Besides, in case of one-to-many mapping, the forwarding decision of an upstream DVS should also be 'forwarding to DVS x', where x is the downstream DVS of the virtual switch.

Routing in the Physical Network: SVirt maintains a separate routing system in the physical network, which is irrelevant to the routing in each virtual network. Although the routing system of the physical network is not the focus of this work, in general the routing paths between physical switches can be either centrally configured by SVirt controller using ways like in Portland [21], Spain [22] or OpenFlow, or determined by traditional distributed routing protocols.

C. SVirt Data Plane: Carrying Forwarding Context Information in Packets

Context-loss Problem: Forwarding context in the data plane, *e.g.*, the incoming interface of a packet, is important to assist the forwarding decision in a switch. Since SVirt enables highly flexible mapping from virtual networks to the physical network, it is difficult to obtain the forwarding context of a virtual network directly from the physical network, which we call the "context-loss problem" in a virtual network. Specifically, we should address the following issues.

First, since multiple DVSes from different virtual networks or the same network can reside in one physical switch, we need to identify the DVS to process an incoming packet. We cannot get the information from the packet header generated by a virtual network or from the physical network's forwarding plane.

Second, we need a way to obtain the incoming interface of a packet in the virtual network. Previous solutions [7], [8] install a module in physical switches to maintain the mapping from a physical interface to a virtual interface. However, this solution does not apply in SVirt. Take Fig. 2 as the example. If DVS a receives a packet from physical switch C, the incoming virtual link can be either $c \rightarrow a$ or $d \rightarrow a$. Since the two virtual links are both mapped to the physical link $C \rightarrow A$, a simple interface mapping relationship cannot distinguish the two virtual interfaces. Hence, we cannot infer the incoming virtual interface from the physical network's context.

Third, as aforementioned, in case of one-to-many mapping, the in-pipeline context information of upstream stages, such as the meta_data, action set, will get lost in downstream stages.

To solve the context-loss problem as listed above, which is particularly caused by many-to-one mapping and one-to-many mapping, SVirt explicitly carries all the required forwarding context information of a virtual network in the packet, instead of inferring from the physical network.

SVirt Packet Format: Fig. 7 shows the SVirt packet format. Note that besides the packet forwarding logic in the virtual networks, there is a separate forwarding plane in the physical network which delivers packets from one physical switch to another. We cannot use the packet header generated by a virtual network as the forwarding header in the physical network, due to possible address overlap between different virtual networks. Hence, we add an outside forwarding header which encapsulates the tenant-specific packet header. As in many previous works that isolate virtual networks [21], [23], [24], the outside forwarding header can be either a layer-2 or a layer-3 header, based on whether layer-2 or layer-3 forwarding fabric is used in the physical network. Using the outside forwarding header, SVirt provides full header space for each tenant.

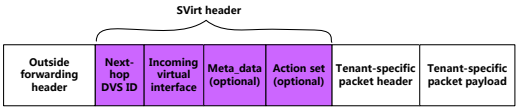


Fig. 7. SVirt packet format.

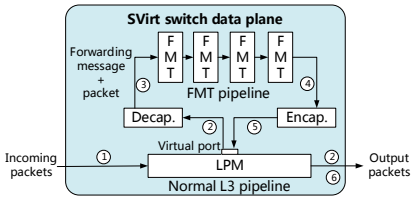


Fig. 8. Packet processing flow in SVirt.

We insert an SVirt header between the tenant-specific packet header and the outside forwarding header, to pack all the required forwarding context information in a virtual network. The SVirt header consists of two required fields, *i.e.*, 32-bit next-hop DVS ID and 32-bit incoming virtual interface for the next-hop DVS, as well as two optional fields, *i.e.*, the meta_data and action set. The two required fields are updated hop-by-hop during the forwarding process, while the two optional fields will be carried only between DVSes that are of the same virtual switch.

Packet Processing Flow: Fig. 8 illustrates the packet processing flow. When a physical switch receives an incoming packet, it checks whether it is the packet's destination by inspecting the outside forwarding header. If so, it decapsulates the outside forwarding header, and then uses the forwarding context information from the SVirt header and the tenant-specific packet header as the input to the late-binding key extractor; if not, it simply forwards the packet out by the physical network's forwarding plane.

After executing the forwarding logic of the virtual network by extracting the late-binding matching keys and looking up the flow tables, the switch encapsulates the updated tenant-specific packet header by an updated SVirt header and outside forwarding header. For the SVirt header, the switch fills the next-hop DVS field as indicated by the 'forwarding to DVS x' instruction pre-installed by SVirt controller, and fills the incoming virtual interface field as the interface ID towards the next-hop virtual switch, which is obtained from the pre-installed DVS neighboring table. If the next-hop DVS belongs

to the same virtual switch as the processing DVS, the incoming virtual interface field remains the same as received from last hop, and the meta_data field and action set field are inserted as output by the last pipeline stage of the processing DVS. Next, the switch looks up the DVS neighboring table again to locate the physical switch that the next-hop DVS resides in, and fills the outside forwarding header based on the information. Finally, the switch forwards the packet out by the physical network's forwarding plane. It is notable that if the the next-hop DVS resides in the same physical switch as the processing one, which may occur due to many-to-one mapping, the packet will be output to the loopback interface of the switch.

IV. Simulations

We write an event-driven simulator to evaluate the advantage of SVirt.

A. Simulation Setup

Methodology: To check the contribution of each technical feature in SVirt, we compare SVirt with three other possible mechanisms, namely, SVirt without late-binding key extractor (SVirt w/o LBX), SVirt without many-to-one mapping (SVirt w/o M2O) and SVirt without one-to-many mapping (SVirt w/o O2M). Without late-binding key extractor, we use the full packet header plus the matching field bitmap as the matching key in a pipeline stage, which results in more TCAM occupation and accordingly less accepted virtual networks; without many-to-one mapping, large virtual network requests with more virtual switches than the available physical switches will be rejected; without one-to-many mapping, virtual network requests with too high TCAM space requirement on an individual virtual switch will be rejected. Note that none of previous SDN virtualization solutions [5], [6], [7], [8] is better than any of the four mechanisms we use in the evaluation, since they do not support any of the three features at all.

Setting of Physical Network: In most of the simulations, we use Fat-Tree [26] as the topology of the physical network. The physical switches have 32 ports and the total number of switches is 1280. Each physical switch is equipped with a four-stage processing pipeline. Similar with [10], the matching table of each pipeline stage has 16 TCAM blocks, each one with 40-bit width and 2k entries. As discussed in Section III-A, the width of a matching table is reconfigurable when there is no active forwarding entry. Besides Fat-Tree topology, we also use Jellyfish [27] when evaluating the virtual link stretch.

Setting of Virtual Networks: We use time window to model the dynamical arrival and leaving of virtual networks, as used in [13]. The arrival of virtual networks follows the poisson distribution, with the default expected arriving rate as 32. Each virtual network is expected to stay for three time windows. The numbers of virtual switches in virtual networks, the pipeline lengths (number of pipeline stages) of the virtual switches, as well as the TCAM capacities (Kb) requested by virtual pipeline stages, all follow random distribution, with the default expected values of 240, 4 and 1280, respectively. We also vary the expected values of these parameters and study their impacts. When generating the topology of a virtual network, the possibility of a link between any pair of virtual switches is set as 0.5. Our simulation runs for 500 time windows.

B. Accumulated Accepted Virtual Networks Over Time

We set the expected values of all the parameters of the virtual network requests as the default, and count the accumulated number of accepted virtual networks over the simulation time. Fig. 9 shows the result. We can observe that each of the three features in SVirt has considerable contribution to the total number of accepted virtual network requests, which is directly translated to the revenue of the cloud provider. Late-binding key extractor is particularly important in saving the TCAM space of physical switches and improving the capability of the physical network. During the simulation time, SVirt will lose 87.3%, 58.2% and 51.4% tenants if not using late-binding key extractor, many-to-one mapping and one-to-many mapping, respectively.

Since the virtual networks have different sizes and different TCAM space requirements, it is arguable that the number of virtual networks only is insufficient to reflect the physical network's capacity in accepting virtual networks. In what follows, we use the total TCAM capacity of all the virtual switches of accepted virtual networks during the simulation time as the evaluation metric.

C. Impact of Virtual Networks' Arrival Rate

We vary the expected arrival rate of virtual networks from 16 to 44 in the poisson distribution, and set the expected values of all the other parameters as default. We rerun the simulation for each expected arrival rate, and the result is shown in Fig. 10. With the increase of virtual networks' arrival rate, the total TCAM capacity of accepted virtual networks quickly grows, and then stays relatively steady due to physical network's resource constraint. Lack of any of the three technical features in SVirt will significantly degrade the capability of the cloud. Moreover, we find that when the cloud is busy, the total TCAM capacity of accepted virtual networks is very close to (over 92%) the total TCAM space of the physical network, which indicates that our virtual network allocation algorithm successfully finds the best-matching physical switches to serve the virtual switches' pipelines.

D. Impact of Virtual Network Size

Fig. 11 shows the impact of virtual network size by varying the expected number of virtual switches in a virtual network from 120 to 400, and the expected values of all the other parameters are set as default. Note there are 1280 switches in the physical network. We can observe that except SVirt w/o M2O, the virtual network size has little impact on the other three mechanisms, because the many-to-one mapping can naturally embrace large virtual networks with more virtual switches than the available physical switches. However, without many-to-one mapping, the total TCAM capacity of accepted virtual networks decreases with larger virtual network size, when the expected size is larger than 280. It is because in

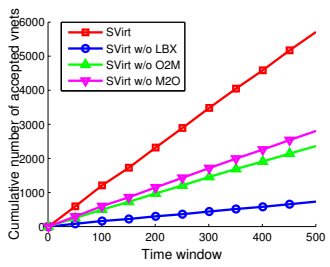


Fig. 9. Accumulated number of accepted virtual networks over time.

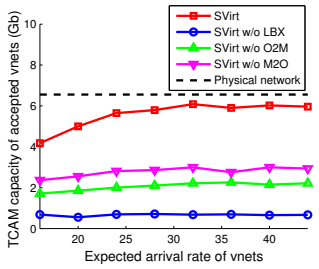


Fig. 10. TCAM capacity of accepted virtual networks over requesting rate.

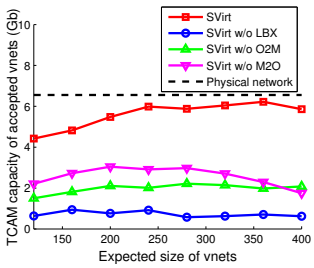


Fig. 11. TCAM capacity of accepted virtual networks over vnet size.

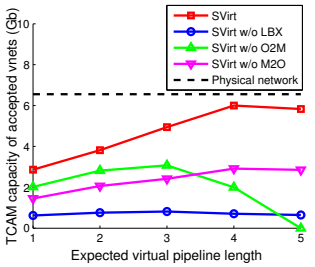


Fig. 12. TCAM capacity of accepted virtual networks over pipeline length.

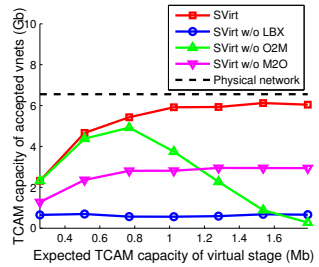


Fig. 13. TCAM capacity of accepted virtual networks over stage capacity.

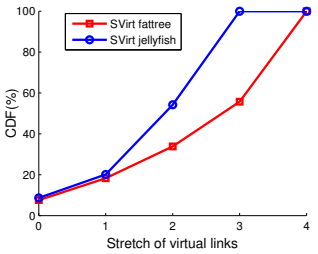


Fig. 14. Cumulative distribution function of virtual links' stretches.

this case, more large requests are likely to exceed the residual number of physical switches and thus get rejected. When the expected virtual network size is 400, the total TCAM capacity of accepted virtual networks drops by about 30% compared with when the expected size is 280.

E. Impact of Virtual Pipelines

SVirt supports flexible specification of the virtual pipelines. We vary the expected virtual pipeline length from 1 to 5 and set the expected values of other parameters as default. The result is shown in Fig. 12. If there is no one-to-many mapping (SVirt w/o O2M), the total TCAM capacity of accepted virtual networks will degrade significantly when the expected virtual pipeline length is larger than 3, because it is more difficult to find a physical switch to host the long virtual pipelines. On the contrary, for the other three mechanisms with one-to-many mapping, the growth of virtual pipeline length results in higher accepted TCAM capacity, since more TCAM resources are requested and a virtual network will not be rejected simply due to its long virtual pipeline length. Note that in SVirt w/o LBX, the physical network's capacity is exhausted even when the expected virtual pipeline is 1, due to too much TCAM space occupied by full key matching; as a result, there is no obvious change of the accepted TCAM capacity with longer virtual pipelines.

We also vary the expected TCAM capacity requested by a virtual pipeline stage from 256 Kb to 1.8 Mb, with the expected values of other parameters set as default. In this setting, the expected TCAM capacity of a requesting virtual pipeline stage is 0.4 to 2.8 times that of a physical pipeline stage. The result is shown in Fig. 13. Similarly, without one-to-many mapping, the total TCAM capacity of accepted virtual networks significantly degrades when the expected TCAM capacity of a virtual stage exceeds 768 Kb. It is because if we disable one-to-many mapping, the fragmentation of free TCAM space is less likely to be used by virtual networks with large virtual pipeline stage. Similar as in Fig. 12, the other three mechanisms with one-to-many mapping enjoy growing TCAM capacity of accepted virtual network when the size of virtual stage increases.

Both Fig. 12 and Fig. 13 demonstrate that one-to-many mapping is important in helping SVirt accept virtual switches with long virtual pipeline or large TCAM space requirements.

F. Virtual Link Stretch

SVirt flexibly maps virtual switches to physical switches with the purpose of minimizing the wasted TCAM space. If two neighboring virtual switches are allocated to non-adjacent physical switches, their distance (in terms of hops) in the physical network will be more than one. We define *virtual link stretch* as the distance of a virtual link in the physical network. We run the simulation in both Fat-Tree and Jellyfish networks, each with 1280 switches. Fat-Tree represents the cloud network with regular topology, while Jellyfish represents that with incrementally-expanding topology. The expected values for all the parameters of virtual network requests are set as default.

Fig. 14 shows the cumulative distribution function of the virtual link stretch in the two networks. About 20% of the virtual links have a stretch of 0 or 1. A stretch of 0 means that two ends of the virtual link are put to the same physical switch. In Jellyfish, no virtual link is put to two physical switches with more than 3 hops away; while in Fat-Tree, about 45% of the virtual links span four physical hops. The reason is that each pod of the Fat-Tree network only consists of 32 physical switches, while the expected virtual network size is

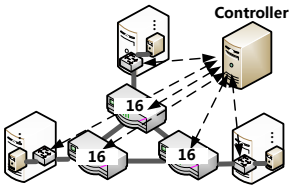


Fig. 15. Physical testbed.

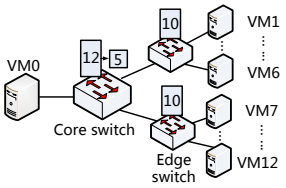
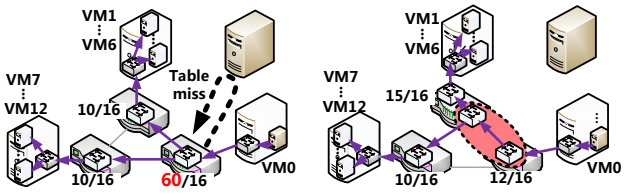


Fig. 16. Virtual network.



(a) Traditional mapping. (b) SVirt mapping.

Fig. 17. Mapping the virtual switches to the physical switches. x/y denotes that the switch has y TCAM entries while the virtual network placement requires at most x TCAM entries in the switch.

240. Hence, it's common that two neighboring virtual switches within a virtual network are placed in different pods of the network since we seek to minimize the wasted TCAM space in switches. In this case, the virtual link crosses 4 hops in the physical network. On the contrary, the Jellyfish topology has abundant paths between switches and the average distance between switches is much shorter than Fat-Tree [27].

V. Prototype based Experiment

In this section we carry out prototype-based experiment to evaluate the performance of SVirt in real systems.

A. Experiment Setup

NetFPGA Implementation of SVirt Switch: We develop a SVirt switch prototype on 1Gb NetFPGA platform base on the reference implementation in [28]. We successfully use two time cycles to realize the late-binding key extractor, one for getting the matching field bitmap and one for extracting the key from the selection network. As in the reference implementation, the processing pipeline contains only one stage due to resource limitation. But with one-to-many mapping, it is easy to realize the functionality of a virtual SDN switch with multi-stage pipeline by multiple NetFPGA cards. It is worth noting that the reference implementation consumes 43% of the slice flip flops and 72% of the LUTs (look up tables) in the NetFPGA board, while we add only 0.4% of the slice flip flops and 2% of the LUTs to support SVirt. The implementation comprises of about 4k lines of Verilog codes. We plan to release the source code to the community later.

Physical Testbed: We setup a small testbed to run SVirt. As illustrated by Fig. 15, the testbed consists of three NetFPGA prototype SVirt switches and four DELL PowerEdge R210 servers. Each NetFPGA SVirt switch uses three 1Gb ports to forward traffic and one configuration port to communicate with the SVirt controller. We set the TCAM capacity of a SVirt switch as 16 entries. The servers are equipped with four 3.1 GHz CPU cores, 8 GB memory and a 1Gb NIC port on each. Three servers host the VMs for the tenants while the fourth server hosts the SVirt controller as well as the tenant-specific SDN controller. We deploy OpenvSwitch [25] as the in-server virtual switch and modifies it to support SVirt as well.

Virtual SDN Request: We emulate a virtual SDN request as shown by Fig. 16. The virtual network has 3 virtual switches and 13 VMs connected as the topology in the figure. The core virtual switch has a two-stage processing pipeline. The first stage matches the destination IP address and determines the forwarding interface; while the second stage matches the destination TCP port to decide whether to drop the packet. The two stages require 12 and 5 TCAM entries respectively. It implies that the virtual network can configure the forwarding states for at most 60 flows (with disjoint destination IP addresses and TCP ports) crossing the core virtual switch. The two edge virtual switches make a single-stage processing of NAT translation, requiring 10 TCAM entries on each.

Comparison: In the experiment we compare SVirt with a traditional SDN virtualization approach. Fig. 17(a) demonstrates how to map the virtual SDN request to the physical network by the traditional approach. Given that the NetFPGA based SVirt switches have only one pipeline stage and the traditional approach supports neither many-to-one mapping nor one-to-many mapping, the two-stage pipeline in the core virtual switch has to be transformed to a single stage. As a result, the number of forwarding entries will be the Cartesian product of the two stages. When more than 16 flows cross the core virtual switch, the switch's TCAM space will be overloaded. The two virtual edge switches are mapped to the other two SVirt switches, respectively.

Contrarily, Fig. 17(b) shows the mapping in SVirt. With one-to-many mapping, SVirt allocates the two-stage core virtual switch as two DVSes on two SVirt switches. With many-to-one mapping and late-binding key extractor, one DVS of the core virtual switch (with 5 flow entries) can share the SVirt switch with one edge virtual switch. Another edge virtual switch is allocated to the third SVirt switch. Therefore, the TCAM requirement of all the virtual SDN switches are satisfied.

B. Results

We generate different numbers of test flows (with disjoint destination IP addresses and TCP ports) from VM 0 to the other 12 VMs in the virtual network, varying from 20 to 60. The size of each flow is set as 100 MB. We run SVirt and the traditional approach separately, and measure the aggregate network throughput, the average flow completion time, as well as the number of flow table misses.

Fig. 18 shows the aggregate throughput of all the flows for the two approaches. Due to the capacity limit of the NIC port in the sever hosting VM 0, the theoretical upper bound of the aggregate throughput in the network is 1 Gbps. From the figure, under all conditions the aggregate throughput of the flows in SVirt is around 920 Mbps, which is close to the upper bound considering the overhead of the SVirt header and the outside forwarding header added in every packet. On the contrary, the aggregate throughput in the traditional approach is always

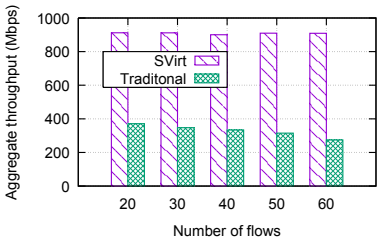


Fig. 18. Aggregate throughput of flows.

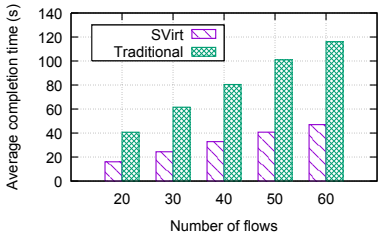


Fig. 19. Average flow completion time.

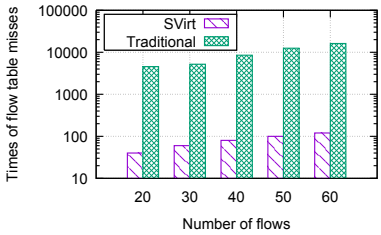


Fig. 20. Times of flow table misses.

under 400 Mbps. When the number of flows increases from 20 to 60, the aggregate throughput in the traditional approach decreases from 371 Mbps to 274 Mbps. In the case of 60 flows, the aggregate throughput of SVirt is over three times that of the traditional approach. Fig. 19 demonstrates the average flow completion time, which increases with more flows due to the bandwidth competition among them. In general, the flows enjoy much shorter completion time in SVirt than in the traditional approach.

The fundamental reason for the performance gap between SVirt and the traditional approach comes from different capabilities of the two mapping methods to support flows' forwarding states. In the traditional approach, when there are more than 16 flows, the TCAM space in the SVirt switch which hosts the core virtual switch is exhausted and the table miss event occurs. Upon table miss, the packet is delivered to the tenant-specific SDN controller, which then randomly deletes a forwarding entry in the switch and installs a new entry for the requesting flow. Consequently, a table miss event causes much larger delay for the packets of new flow and makes more running flows interrupted, which significantly hurts the network throughput and postpones the flow completion time. But SVirt can support at most 60 flows in the two-stage core virtual switch without forwarding entry replacement. Fig. 20 illustrates the times of flow table misses during the whole experiment period for all the flows. SVirt has only a few hundred times of table misses for setting up entries for new flows, while the traditional approach experiences two orders more table misses.

VI. Conclusion

We design SVirt, a substrate-agnostic SDN virtualization architecture for multi-tenant cloud. SVirt enables tenants to request a virtual network with arbitrary topology, arbitrary number of switches, as well as arbitrary processing pipeline and TCAM space in an individual switch. SVirt achieves the goal by redesigning physical switch's processing pipeline to determine the matching keys of packets in a late-binding way, supporting many-to-one and one-to-many mapping in the control plane, and explicitly carrying the forwarding context information in the packet header to overcome the context-loss problem. We have developed a prototype of SVirt switch by NetFPGA and plan to release the source code to the community later. Evaluation results show that, compared with traditional approaches, SVirt significantly enhances the cloud's capability to accept various virtual SDN requests and improves network throughput.

References

[1] SDN Strategies (2014 Edition), Annual North America Enterprise Survey, https://www.infonetics.com/cgp/lp.asp?id=863.

[2] OpenFlow 1.4 spec, http://www.opennetworking.org/

[3] Broadcom StrataXGS Trident II, http://www.broadcom.com/collateral/pb/56850-PB03-R.pdf

[4] P. Kazemian, G. Varghese and N. McKeown, "Header Space Analysis: Static Checking For Networks", *USENIX NSDI'12*.

[5] R. Sherwood, G. Gibb, K. Yap, *etc.*, "Flowvisor: A Network Virtualization Layer", *OPENFLOW-TR-2009-1*, 2009.

[6] D. Drutskoy, E. Keller, J. Rexford, *etc.*, "Scalable Network Virtualization in Software-defined Networks", *IEEE Internet Computing*, 2013.

[7] A. AI-Shabibi, M. Leenheer, M. Gerola, *etc.*, "OpenVirteX: Make Your Virtual SDNs Programmable", *ACM HotSDN'14*.

[8] R. Doriguzzi, M. Gerola, R. Riggio, *etc.*, "VeRTIGO: Network Virtualization and Beyond", *IEEE EWSDN' 12*.

[9] H. Pan, H. Guan, J. Liu, *etc.*, "The FlowAdapter: Enable Flexible Multi-Table Processing on Legacy Hardware", *ACM HotSDN'13*.

[10] P. Bosshart, G. Gibb, H. Kim, *etc.*, "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN", *ACM SIGCOMM'13*.

[11] H. Song, "Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane", *ACM HotSDN'13*.

[12] A. Fischer, J. Botero, M. Till Beck, *etc.*, "Virtual Network Embedding A Survey." *IEEE Communications Surveys & Tutorials*, 2013.

[13] M. Yu, Y. Yi, J. Rexford, *etc.*, "Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration", *ACM SIGCOMM CCR*, 2008.

[14] M. Chowdhury, R. Muntasir and B. Raouf, "Virtual Network Embedding with Coordinated Node and Link Mapping", *IEEE INFOCOM'09*.

[15] X. Cheng, S. Su, Z. Zhang, *etc.*, "Virtual Network Embedding Through Topology-aware Node Ranking", *ACM SIGCOMM CCR*, 2011.

[16] Intel Ethernet Switch FM6000 Series - Software Defined Networking, http://goo.gl/djEdq4

[17] M. Casado, T. Koponen, R. Ramanathan, *etc.*, "Virtualizing the Network Forwarding Plane", *ACM PRESTO'10*.

[18] M. Yu, J. Rexford, M. Freedman, *etc.*, "Scalable Flow-based Networking with DIFANE", *ACM SIGCOMM'10*.

[19] N. Kang, Z. Liu, J. Rexford, *etc.*, "Optimizing the 'One Big Switch' Abstraction in Software-Defined Networks", *ACM CoNEXT'13*.

[20] Y. Kanizo, D. Hay, K. Issac, "Palette: Distributing Tables in Software-defined Networks", *IEEE INFOCOM'13*.

[21] R. Niranjan, A. Pamboris, N. Farrington, *etc.*, "Portland: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric", *ACM SIGCOMM'09*.

[22] J. Mudigonda, P. Yalagandula, M. AI-Fares, *etc.*, "SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies", *USENIX NSDI'10*.

[23] A. Greenberg, J. Hamilton, N. Jain, *etc.*, "VL2: A Scalable and Flexible Data Center Network", *ACM SIGCOMM'09*.

[24] J. Mudigonda, P. Yalagandula, J. Mogul, *etc.*, "Netlord: a Scalable Multi-tenant Network Architecture for Virtualized Datacenters", *ACM SIGCOMM'11*.

[25] B. Pfaff, J. Pettit, T. Koponen, *etc.*, "The Design and Implementation of Open vSwitch", *USENIX NSDI'15*.

[26] M. Al-Fares, A. Loukissas, A. Vahdat, *etc.*, "A Scalable, Commodity Data Center Network Architecture", *ACM SIGCOMM'08.*

[27] A. Singla, C. Hong, L. Popa, *etc.*, "Jellyfish: Networking Data Centers Randomly", *USENIX NSDI'12.*

[28] J. Naous, D. Erickson, G. Covington, *etc.*, "Implementing An OpenFlow Switch on the NetFPGA Platform", *ACM ANCS'08.*