PVFlow: Flow-Table Virtualization in POF-based vSDN Hypervisor (PVX)

Shengru Li, Kai Han, Huibai Huang, Zuqing Zhu†
School of Information Science and Technology, University of Science and Technology of China, Hefei, China
†Email: {zqzhu}@ieee.org

***Abstract*—It is known that by combining network virtualization with software-defined network (SDN), people can realize virtual SDNs (vSDNs) for enhanced programmability, adaptivity and cost-effectiveness. Meanwhile, protocol-oblivious forwarding (POF) can overcome the protocol-dependent restriction of OpenFlow and provide a more generic abstraction model of the forwarding elements in SDNs to realize a protocol-independent data plane. In this work, we design a flow-table virtualization module, *i.e.*, PVFlow, to realize resource-efficient flow-table virtualization for POF-based vSDNs. To fully explore the flexibility of POF and support arbitrary matching fields in vSDNs' flow-tables, we propose the idea of "BigMatch", which enables different flow-tables in vSDNs to share a matching stage in a substrate POF switch efficiently. We implement PVFlow in a POF-based network virtualization hypervisor (NVH) system, *i.e.*, PVX, and verify its effectiveness on improving the efficiency of flow-table virtualization with experiments.**

***Index Terms*—Software-defined networking (SDN), Protocol-oblivious forwarding (POF), Network virtualization hypervisor, Flow-table virtualization.**

I. Introduction

The fast development of Internet has generated tremendous new applications with various quality-of-service (QoS) demands. To adapt to them, Internet service providers (ISPs) need a comprehensive solution, which should not only properly address the ossification of current Internet infrastructure [1] but also provide sufficient network programmability to satisfy the unique requirements of applications. The dilemma of ossified Internet infrastructure can be resolved with network virtualization [2, 3], which builds multiple virtual networks (VNs) over a shared substrate network (SN) and isolates resources for the VNs to let them customize their own network environments. However, the traditional network virtualization approaches, *e.g.*, VLAN, VXLAN, and GRE, only facilitate resource virtualization and isolation, but cannot provide the tenants a powerful network control and management (NC&M) scheme that can be used to satisfy their unique QoS requirements. The centralized NC&M scheme in software-defined network (SDN) [4] can fulfill this missing puzzle piece. When network virtualization meets SDN to realize virtual SDNs (vSDNs), the advantages such as enhanced programmability and adaptivity can be realized [5, 6].

The network virtualization hypervisor (NVH) is a key element for building vSDNs. Specifically, similar to the IT virtualization hypervisor that allocates physical IT resources to virtual machines, an NVH slices the network resources (*e.g.*, bandwidth and flow-tables) in an SN for vSDNs [5]. Previously, to build OpenFlow-based vSDNs, people have developed software systems such as FlowVisor [7] and OpenVirteX (OVX) [8]. Nevertheless, it is known that the protocol-dependent nature of OpenFlow restricts the programmability of network virtualization systems [9–11]. Specifically, an OpenFlow switch has to know the protocol headers to parse and match to the fields in them, which could cause compatibility issues if a vSDN needs to support protocols that have not been standardized in the OpenFlow specifications. Therefore, a protocol-independent NVH would be desired to realize the future-proof feature such that vSDNs can be programmed to support new protocols seamlessly and timely [11].

Recently, the idea of protocol-independent forwarding (PIF) [12] has been put forward by the Open Networking Foundation (ONF), and for its practice, the programming protocol-independent packet processors (P4) [10] and the protocol-oblivious forwarding (POF) [13] are the pioneers. POF provides a more generic abstraction model of the forwarding elements, which locates the data fields in packets through <*offset, length*> tuples without a protocol parser. Here, *offset* denotes the start bit-location of a field in a packet while *length* represents its length in bits. Nevertheless, due to the protocol-independent nature of POF, realizing an NVH based on it is more challenging, especially for the flow-table virtualization that isolates the storage resources in substrate switches to store the virtual flow-tables of each vSDN. This is because POF switches literally support arbitrary matching fields in their flow-tables instead of few predefined ones, and thus efficiently organizing the flow-tables of vSDNs in substrate switches would not be an easy task to do.

In this work, we extend the POF-based NVH (*i.e.*, PVX) that we developed in [11] and design a flow-table virtualization module, namely, PVFlow, to realize resource-efficient flow-table virtualization for POF-based vSDNs. The rest of the paper is organized as follows. Section II briefly introduces the background of POF-based flow-table virtualization. The system design and operation principle of PVFlow are described in Section III. Then, we present the implementation of PVFlow in Section IV, and the experimental evaluation is discussed in Section V. Finally, Section VI summarizes the paper.

II. Background and Motivations

Different from the OpenFlow switches that operate based on pre-defined matching fields, POF switches enable the

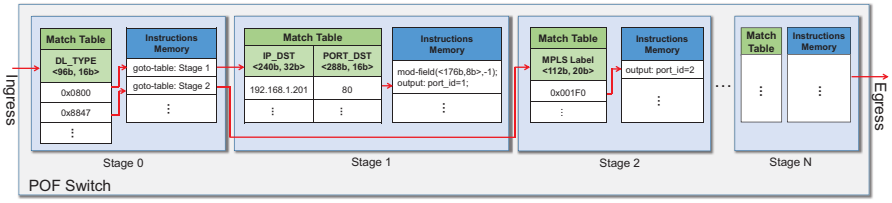

Fig. 1. Example on the organization of flow-tables in a POF switch.

network operator to customize the matching fields in flow-tables arbitrarily by using the <*offset, length*> tuples [9, 14] and a generic forwarding instruction set (*i.e.* POF-FIS) [15]. Specifically, a controller can create a flow-table in a POF switch by specifying the table ID, table size, and matching fields with the *TableMod* message defined in the POF protocol.

For instance, Fig. 1 gives an example on how the flow-tables are organized in a POF switch. Here, the POF switch has N matching stages, each of which includes a match table that is stored in the ternary content addressable memory (TCAM) and an instruction block in the normal memory. Specifically, each entry in the match table corresponds to the matching field(s) of an entry in a flow-table, and it contains a pointer to an entry in the corresponding instruction block. Then, for a packet, if a positive match is obtained by a match table entry, the switch will invoke the corresponding instruction(s) in the instruction block to process it. Note that, since TCAM is usually expensive and power hungry [16], one should always try to save its usage to improve the cost-effectiveness of the network system, which is also our objective in this work.

The switch in Fig. 1 is programmed to realize two packet processing pipelines, *i.e.*, for IPv4/UDP and multi-protocol label switching (MPLS) packets, respectively. Each pipeline consists of two matching stages, whose match tables are all defined with the <*offset, length*> tuples. *Stage* 0 is to match to the *EtherType* field in a packet's Ethernet header, which locates from the 96-th bit with a length of 16 bits and tells the switch whether the Layer-3 protocol is IPv4 (*i.e.*, with *EtherType* as $0x0800$) or MPLS (*i.e.*, with *EtherType* as $0x8847$). Then, if the switch finds that a packet is an IPv4 one, it processes the packet with the match table in *Stage* 1, which specifies the IP destination address and UDP port to be matched with and the corresponding instructions for a positive match. Specifically, the switch decreases the packet's *TTL* field (*i.e.*, <176 bits, 8 bits>) by 1 and outputs it through *Port* 1. Otherwise, the packet processing procedure for MPLS packets is in *Stage* 2.

Note that, to realize vSDNs using different protocols over an SN, each substrate POF switch may need to be programmed to support multiple protocols simultaneously as shown in Fig. 1. As the flow-tables of virtual POF switches can contain arbitrary matching fields with various lengths, the flow-table virtualization to organize them in the TCAM of substrate switches is essential and can greatly affect the efficiency of vSDN slicing. Previously, in [11], we developed a POF-based NVH, namely, PVX, which explored the programmability of POF-FIS to realize protocol-independent vSDNs. However, we did not try to optimize the flow-table virtualization in it. Specifically, PVX allocates the whole TCAM of a matching stage to a virtual switch for creating a flow-table for it, even though the virtual switch's flow-table will not contain many flow-entries, *i.e.*, the matching stages in a substrate switch could not be shared by different virtual flow-tables. This is because the matching fields of different flow-tables would usually not be the same as those in *Stage* 0 in Fig. 1, due to the flexibility of supporting arbitrary matching fields. Therefore, it is not possible to merge them in one matching stage without proper preprocessing, *i.e.*, the flow-tables for IPv4/UDP and MPLS packets in Fig. 1 need to occupy two matching stages. Apparently, this type of flow-table virtualization can waste a lot of TCAM in substrate switches.

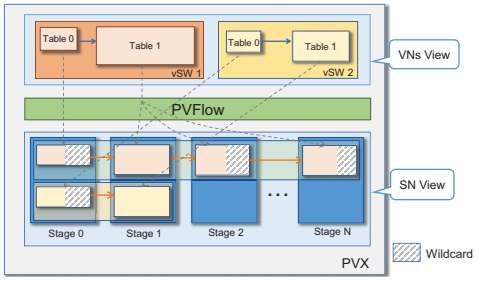

Fig. 2. Principle of forming BigMatches with PVFlow.

III. Operation Principle

A. Transforming Matching Fields into BigMatches

To address the aforementioned issue, we design a new software module, namely, PVFlow, to greatly improve the resource efficiency of the flow-table virtualization in PVX. First of all, we propose an approach to unify the flow-tables that contain arbitrary matching fields with various lengths, and enable them to share the TCAM of a matching stage in a substrate switch. Specifically, we transform the matching field(s) in each flow-table into a generic matching field, namely, "BigMatch". For example, for a flow-table that contains two matching fields as <240 bits, 32 bits> and <288 bits, 16 bits>, we combine them into a BigMatch as <240 bits, 64 bits> (*i.e.*, starting from the 240-th bit and covering 64 bits), and set the bits for <272 bits, 16 bits> as wildcards since they do not need to be matched according to the two original matching fields. Meanwhile, the TCAM in substrate switches is divided into matching stages with fixed-length entries too. Then, each BigMatch can be stored as an entry in a matching stage.

Note that, in most cases, the length of a BigMatch would not be exactly the same as that of an entry in the matching stage. This issue can be resolved as follows. If the length of the BigMatch is shorter, we stuff more wildcards at the end of it to make them in equal length. For example, we need to stuff 32 wildcard bits at the end of the aforementioned BigMatch <240 bits, 64 bits> to store it in a 96-bit matching stage entry. Otherwise, if the BigMatch is too long for a matching stage entry, we can divide it into multiple fragments to store in multiple stages. Here, to save TCAM, if the start bit-locations of the original matching fields are separated by a length that is comparable to the length of a matching stage entry, we have the option of not combining them into one BigMatch but transforming each of them into a BigMatch.

Fig. 2 explains the principle of forming BigMatches intuitively. Here, we consider two virtual switches (*i.e.*, *vSWs* 1 and 2), and both of them contain two flow-tables. Since the lengths of the BigMatches obtained from the first flow-tables in *vSWs* 1 and 2 are shorter than that of a matching stage entry in the substrate switch, the BigMatches (*i.e.*, with certain stuffing wildcards) can both be stored in *Stage* 0. Then, for *Table* 1 in *vSW* 1, we assume that it consists of three matching fields and generate two BigMatches out of them. The first two matching fields are converted into a BigMatch whose length is longer than a matching stage entry, and thus the BigMatch is divided and stored in *Stages* 1 and 2. Then, since the start bit-locations of the first and third matching fields in *Table* 1 are separated by a length that is comparable to the length of a matching stage entry, we just generate a BigMatch for the third flow-table and store it in *Stage N*. For *Table* 1 in *vSW* 2, the same procedure is applied, and since the length of the BigMatch generated for it is the same as that of a matching stage entry, it is stored in *Stage* 1.

B. Allocating TCAM Space to Store BigMatches

When a substrate switch connects to PVX, PVFlow initializes all the matching stages in it with *TableMod* messages. Then, during the run-time of a vSDN, its controller can create or scale-out the flow-tables based on BigMatches in its virtual switches also with *TableMod* messages. Specifically, PVFlow calculates the required TCAM space based on the information in such a message (*e.g.*, flow-table size, number of matching fields in each flow-entry, and total length of each flow-entry) and the current matching stage usage of the vSDN, and then allocates a block of matching stage with enough space for the vSDN exclusively in the specified substrate switch. The block of matching stage is allocated based on the first-fit scenario.

C. Translating Flow-tables for vSDNs

After a virtual flow-table has been allocated to one or more matching stages in a substrate switch, the controller of the vSDN can install flow-entries in it. Fig. 3 gives an example on installation and translation of the flow-entries. Here, we assume that *Stage N* is a matching stage whose match table is shared by three tenants (*i.e.*, three vSDNs). To operate the three vSDNs, PVFlow extracts all the matching fields' values in each *FlowMod* message from a vSDN controller, and translates them into the value(s) of the corresponding BigMatch(es). Specifically, as shown in Fig. 3, the flow-entry from *Tenant* 1's controller tries to match to the IPv4 destination address and UDP port of a packet, and PVFlow translates the flow-entry into a 96-bit BigMatch to store in the space allocated to *Tenant* 1 in the substrate switch. Then, a *FlowMod* message that can convey the BigMatch is built by PVFlow and sent to the substrate switch. For the flow-entries from other tenants' controllers, the same translation procedure is applied, and since the BigMatches have the same length and thus can be stored in the same matching stage, the TCAM space in the substrate switch can be utilized more wisely. Meanwhile, we hope to point out that for the BigMatches stored in the same matching stage for different vSDNs, their start bit-locations do not need to be the same. Actually, to point to the right start bit-location for each BigMatch, PVFlow inserts a *MovePacketOffset* instruction in the instruction block for it, which can correctly adjust the offset for the BigMatch.

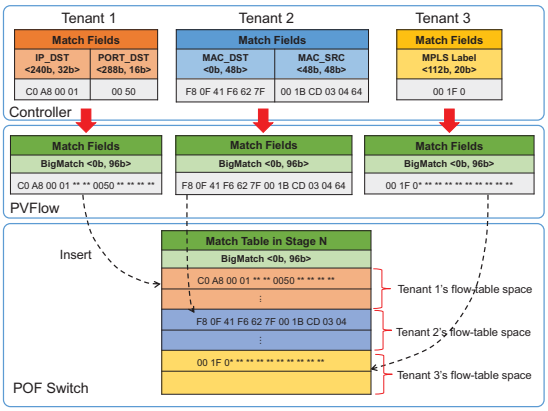

Fig. 3. Example of flow-entry translation.

All these operations are realized with the instructions in POF-FIS and made transparent to the tenants. Therefore, our design of PVFlow does not require any modifications in either the POF controller or the POF switch, which means that it works smoothly with the existing POF controller [9] and switches (*i.e.*, both the open-source software-based switches [17] and the commercial hardware switches [9]).

IV. System Implementation

We implement PVFlow in the PVX developed in our previous work [11]. Fig. 4 shows the functional modules of PVFlow, which are explained as follows.

- **Table Store**: It maintains all the mapping between the tenants' virtual flow-tables and the match tables in matching stages, and also records the availability of the matching stages in substrate switches' TCAM.
- **Table Initializer**: It interacts with substrate switches to create the instances for their matching stages in the Table Store, and initializes the management and BigMatch-based flow-tables in the substrate switches.

- **Table Dispatcher**: It receives the table-creating tasks parsed from the *TableMod* messages from a tenant's controller. Upon receiving such a task, it extracts the information regarding the virtual flow-table and tries to allocate a block of matching stage in the substrate switch. If the virtual flow-table can be created successfully, it records the mapping result in the Table Store. Otherwise, it returns a failure message to the tenant's controller.
- **Flow Translator**: It processes the *FlowMod* messages from the vSDN controllers and checks the Table Store to translate them into the *FlowMod* messages that can be understood by the substrate switches.

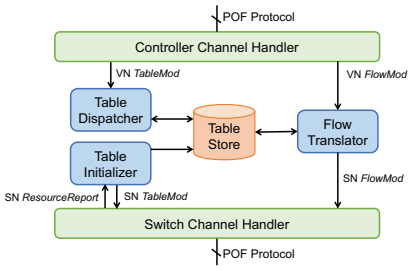

Fig. 4. Functional modules in PVFlow.

V. Performance Evaluation

A. Flow-Table Virtualization Efficiency

Since the major advantage of PVFlow over the existing flow-table virtualization scheme in PVX is that it can significantly improve the TCAM utilization efficiency in substrate switches, we first design an experiment to compare the two approaches on the numbers of provisioned virtual flow-tables on a substrate switch. In our experimental testbed, the POF controller is implemented by extending the POX platform [9] and running it on a Linux server, while each substrate POF switch is realized by running our self-developed software switch on a high-performance Linux server (*i.e.*, Lenovo RD540 equipped with 2.0 GHz 6-core Intel Xeon E5-2620 CPU and 64 GB memory). Note that, since there is no on-the-shelf hardware POF switch at this moment, we have to use the software switches to emulate the behaviors of hardware switches. Specifically, to emulate the limited TCAM on each hardware switch, we apply an upper-bound on the size of the memory that can be used to store virtual flow-tables.

In the experiments, the maximum number of flow-entries that can be stored in each substrate switch is set as 2000. Then, we use PVX to create a vSDN slice, let the controller of the vSDN to install various flow-tables in its virtual switches, and compare PVX with PVFlow (PVX w/ PVFlow) with the one without it (PVX w/o PVFlow). Specifically, the controller encapsulates the virtual flow-table requests in *TableMod* messages, chooses the matching fields in each flow-entry of a virtual flow-table randomly from the combinations of common protocol-fields (*e.g.*, Layer-2, Layer-2/MPLS, Layer-3, and Layer-3/Layer-4 fields), and set the required number of flow-entries in each virtual flow-table as uniformly distributed in $[50, 1000]$. In each experiment, the controller will generate 150 virtual flow-table requests and send them to PVX, and we record how many of them can be provisioned successfully.

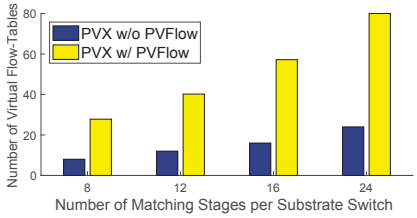

Fig. 5. Results on number of provisioned virtual flow-tables.

Fig. 5 shows the experimental results on the number of provisioned virtual flow-tables in a substrate switch. Here, we consider the situations in which each substrate switch contains different numbers of matching stages. Apparently, PVX w/ PVFlow can effectively improve the flow-table virtualization efficiency as it can provision much more virtual flow-tables with the same substrate resources. Without PVFlow, PVX can only allocate a whole matching stage in the substrate switch to a virtual flow-table, and thus the maximum number of provisioned virtual flow-tables is equal to the number of matching stages in each substrate switch. This, however, would waste a lot of TCAM resources in substrate switches as a virtual flow-table may not occupy the whole space of a matching stage. On the other hand, PVFlow ensures that each matching stage can be shared efficiently by different virtual flow-tables (*i.e.*, from one or multiple vSDNs). These experimental results verify the effectiveness of PVFlow on improving the efficiency of flow-table virtualization.

B. Processing Latency

Note that, compared with the one w/o PVFlow, PVX w/ PVFlow incorporates more operations to process the flow-table virtualization, which would result in more overhead and longer processing latency. To make sure that the additional processing latency would not cause significant performance degradation on packet processing in the network virtualization system, we design an experiment to measure the control channel latency from a vSDN controller to a substrate switch. Specifically, we extend the well-know SDN controller performance test-tool cbench to let it support the POF protocol. The experiments use the same network testbed to measure the average control channel latency for three NVHs, namely, vSDN slicing with 1) OpenFlow-based OpenVirteX (OVX) [8], 2) PVX w/o PVFlow, and 3) PVX w/ PVFlow.

We make sure that the vSDN's controller runs no other applications except for installing *FlowMod* messages according to the *PacketIn* messages from its virtual switches. Specifically, cbench sends *PacketIn* messages to the controller through the NVH and measures the round-trip time (RTT) from sending out a *PacketIn* to receiving a *FlowMod*. The substrate switches are connected according to the Internet2 NDDI topology [18] that consists of 11 nodes, and to measure the worst case scenario, we assume that the topology of each vSDN is exactly

the same as that of the SN, *i.e.*, each substrate switch supports a virtual switch of each vSDN. We set the length of a flow-entry in a matching stage of a substrate switch as 128 bits.

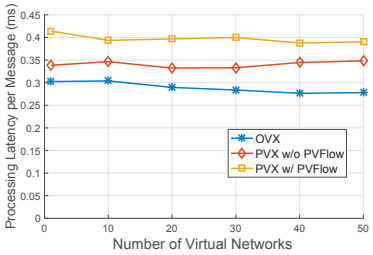

Fig. 6. Results on processing latency.

Fig. 6 shows the experimental results on the control channel latency introduced by OVX, PVX w/o PVFlow and PVX w/ PVFlow. It can be seen that compared with OVX and the one w/o PVFlow, PVX w/ PVFlow only introduce $\sim$0.05 and $\sim$0.1millisecond additional processing latency per message. These results suggest that the processing overhead caused by PVFlow is reasonable and would not cause significant performance degradation on the NVH system.

Finally, we measure the provisioning time in PVX w/ PVFlow, which is defined as the average latency from when the NVH receives a *TableMod* message from a vSDN controller to when the flow-table mapping scheme has been installed in the Table Store of PVFlow. Specifically, each experiment lets the controller generate a batch of *TableMod* messages for creating virtual flow-tables, each of which askes for a flow-table with 100 flow-entries. Fig. 7 illustrates the experimental results for substrate switches with different lengths of matching stages. It can be seen that the provision time decreases with the width of a matching stage in substrate switches. This is because, using matching stages whose flow-entry is too short would lead to frequent flow-table fragmentation when converting the original flow-entries to BigMatches, which introduces additional operation complexity. Therefore, in terms of provision time, setting the width of each matching stage longer would be beneficial. However, a longer width of each matching stage would waste certain TCAM resources in substrate switches, since most of the flow-entries of the vSDNs would normally not be very long. Hence, in practical POF-based NVH systems, we have a tradeoff between the provisioning time and flow-table virtualization efficiency to adjust.

VI. Conclusion

In this paper, we designed a flow-table virtualization module, *i.e.*, PVFlow, to realize resource-efficient flow-table virtualization for POF-based vSDNs. Specifically, to support arbitrary matching fields in vSDNs' flow-tables, we proposed the idea of "BigMatch", which enables different virtual flow-tables to share a matching stage in a substrate switch efficiently. We implemented the proposed PVFlow in a POF-based NVH system, namely, PVX, and conducted experiments to verify its effectiveness on improving the efficiency of flow-table virtualization. The experimental results also confirmed the additional overhead introduced by PVFlow is very small and PVX with it can provision virtual flow-tables quickly.

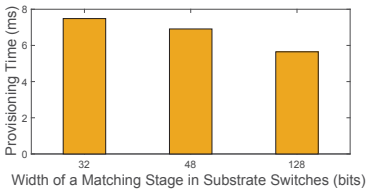

Fig. 7. Results on virtual flow-table provisioning time.

Acknowledgment

This work was supported in part by the NSFC Project 61371117, the Key Research Project of the CAS (QYZDY-SSW-JSC003), and the NGBWMCN Key Project under Grant No. 2017ZX03001019-004.

References

[1] L. Gong, Y. Wen, Z. Zhu, and T. Lee, "Toward profit-seeking virtual network embedding algorithm via global resource capacity," in *Proc. of INFOCOM 2014*, pp. 1–9, Apr. 2014.
[2] L. Gong and Z. Zhu, "Virtual optical network embedding (VONE) over elastic optical networks," *J. Lightw. Technol.*, vol. 32, pp. 450–460, Feb. 2014.
[3] L. Gong *et al.*, "Novel location-constrained virtual network embedding (LC-VNE) algorithms towards integrated node and link mapping," *IEEE/ACM Trans. Netw.*, vol. 24, pp. 3648–3661, Dec. 2016.
[4] Z. Zhu *et al.*, "Demonstration of cooperative resource allocation in an OpenFlow-controlled multidomain and multinational SD-EON testbed," *J. Lightw. Technol.*, vol. 33, pp. 1508–1514, Apr. 2015.
[5] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer, "Survey on network virtualization hypervisors for software defined networking," *IEEE Commun. Surveys Tuts.*, vol. 18, pp. 655–685, First Quarter 2016.
[6] J. Yin *et al.*, "Experimental demonstration of building and operating QoS-aware survivable vSD-EONs with transparent resiliency," *Opt. Express, in Press*, 2017.
[7] R. Sherwood *et al.*, "FlowVisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep.*, pp. 1–13, 2009.
[8] A. Al-Shabibi *et al.*, "OpenVirteX: Make your virtual SDNs programmable," in *Proc. of ACM HotSDN 2014*, pp. 25–30, Aug. 2014.
[9] S. Li *et al.*, "Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability," *IEEE Netw.*, vol. 31, pp. 58–66, Mar./Apr. 2017.
[10] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.
[11] S. Li *et al.*, "SR-PVX: A source routing based network virtualization hypervisor to enable POF-FIS programmability in vSDNs," *IEEE Access*, vol. 5, pp. 7659–7666, Apr. 2017.
[12] OF-PI: A Protocol Independent Layer. [Online]. Available: https://www.opennetworking.org
[13] D. Hu *et al.*, "Flexible flow converging: A systematic case study on forwarding plane programmability of protocol-oblivious forwarding (POF)," *IEEE Access*, vol. 4, pp. 4707–4719, 2016.
[14] S. Li, D. Hu, W. Fang, and Z. Zhu, "Source routing with protocol-oblivious forwarding (POF) to enable efficient e-health data transfers," in *Proc. of ICC 2016*, pp. 1–6, Jun. 2016.
[15] D. Hu *et al.*, "Design and demonstration of SDN-based flexible flow converging with protocol-oblivious forwarding (POF)," in *Proc. of GLOBECOM 2015*, pp. 1–6, Dec. 2015.
[16] D. Kreutz *et al.*, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, pp. 14–76, Jan. 2015.
[17] POFSwitch. [Online]. Available: http://www.poforwarding.org
[18] Internet2 NDDI topology. [Online]. Available: https://www.internet2.edu/media/medialibrary/2013/07/31/Internet2-Network-Infrastructure-Topology.pdf