Towards a Scalable Software-Defined Network Virtualization Platform

Zdravko Bozakov and Panagiotis Papadimitriou

Institute of Communications Technology, Leibniz Universität Hannover, Germany
{zdravko.bozakov, panagiotis.papadimitriou}@ikt.uni-hannover.de

Abstract—Software-defined networking (SDN) has emerged to circumvent the difficulty of introducing new functionality into the network. The widespread adoption of SDN technologies, such as OpenFlow, can facilitate the deployment of novel network functions and new services. Network infrastructure providers can significantly benefit from the SDN paradigm by leasing network slices with SDN support to Service Providers and end-users. Currently, the deployment of arbitrary virtual SDN topologies entails significant configuration overhead for SDN operators.

To this end, we present a SDN virtualization layer that orchestrates the deployment and management of virtual SDNs (vSDN). The so-called SDN hypervisor generates and installs the forwarding entries required for vSDN setup and also coordinates the necessary switch flow table modifications for seamless resource migration. Furthermore, the hypervisor transparently rewrites all control messages enforcing flowspace isolation while giving to the vSDN operator the illusion of exclusive access control. We explore the design space and prerequisites for SDN virtualization, including the selection and encoding of packet identifiers, the resolution of flowspace identifiers, and the configuration and consolidation of multiple virtual flow tables onto a single switch in order to provide support for arbitrary topologies. Furthermore, we discuss the scalability of the SDN control and data plane.

I. INTRODUCTION

Network virtualization constitutes a promising solution for the concurrent deployment and operation of isolated network slices on top of shared network infrastructures [15], [20]. Fundamentally, network virtualization offers significant benefits to Service Providers and network infrastructure providers. In particular, Service Providers (SP) have the ability to efficiently deploy network services within customized virtual networks (VN), which can be based on demand allowing the SP to scale his offered service. Most importantly, network virtualization decouples the network operations from the underlying infrastructure, allowing SPs and VN users to retain the management and control of their own slices. Furthermore, adaptive VN provisioning [8] combined with techniques for virtual resource migration [17] can facilitate VN fault management in response to substrate node/link failures and changes in traffic loads and network topologies. For network infrastructure providers, network virtualization improves resource utilization, reduces operational (OPEX) and technology investment costs (CAPEX), and can generate revenue by leasing VNs to third parties.

Recently, the software-defined networking (SDN) paradigm has emerged to facilitate the deployment of new network functions and services by offering an abstract network view which is decoupled from the underlying switching hardware. SPs can benefit from the faster development cycles associated with the increased abstraction level and, consequently, network infrastructure providers have more incentives to offer virtual networks with SDN support.

FlowVisor [16] has taken a first step towards SDN virtualization. FlowVisor allows multiple users to control SDN slices by regulating the type of flow entries installable by each user, based on the notion of flowspaces. However, FlowVisor provides slicing rather than full virtualization of resources. Hence, it only allows the deployment of vSDN topologies that represent a subset of the substrate network topology. Furthermore, the provision of virtual address spaces is not possible. As such, FlowVisor represents only one building block for the presented SDN virtualization. Besides these restrictions, the planning and deployment of virtual SDNs (vSDN) with FlowVisor requires substantial network operator intervention. In fact, the instantiation of a vSDN, which from a user perspective is virtually indistinguishable from a physical network, is nontrivial. The assignment of virtual resources to physical switches and links, the choice of flowspace, the appropriate traffic encapsulation and flow entries setup require considerable planing and management resources. Furthermore, vSDN deployment entails considerable scalability challenges both for the control and the forwarding plane.

To address these issues, we present a transparent virtualization layer, called SDN hypervisor, that orchestrates the embedding, deployment and management of vSDNs. In particular, the SDN hypervisor: (i) computes the mapping of vSDN topologies, (ii) automates the setup of arbitrary vSDNs by transparently generating required substrate flow entries for packet forwarding and encapsulation, (iii) processes and rewrites control messages, allowing each vSDN operator to configure his own slice as an SDN with exclusive access control, (iv) automates vSDN node and link migration by coordinating the necessary switch flow table updates.

The transparent control message translation enables tenants to install arbitrary packet processing rules within an assigned vSDN, without adversely affecting concurrent users. At the same time, the automation of the infrastructure setup minimizes SDN operator intervention. To scale the SDN hypervisor, we subdivide the SDN substrate into multiple domains and assign a separate controller proxy (CPX) to each domain, based on our previous work [1]. As such, local operations such as flowspace allocation, flow entry installation, and control message translations, are performed independently.
The main goal of this paper is to discuss solutions to some key challenges associated with vSDN setup, management, and control. However, we expect that the parametrization of such a platform will depend on a multitude of factors, such as the network size, topology and utilization, the number of tenants, the vSDN request arrival rates, as well as the lifetime of allocated vSDNs. The evaluation of the SDN hypervisor under different scenarios will be carried out in future work.

The remainder of this paper is organized as follows. Section II discusses the main requirements for SDN virtualization and presents an overview of the proposed SDN hypervisor. In Section III, we delve into flowspace allocation, the encoding of virtual link identifiers, the translation between flow table and virtual link identifiers, and the support for arbitrary vSDN topologies. Section IV presents techniques for vSDN node and link migration. In Section V, we discuss techniques for the scalability of the control and the forwarding plane. In Section VI, we discuss related work. Finally, Section VII highlights our conclusions and gives directions for future work.

II. Virtualizing Software-Defined Networks

Our main goal is to provide support for SDN virtualization while minimizing the associated deployment and operation overhead. As a result, users and SPs may easily request and acquire arbitrary VN topologies from SDN substrate providers on a lease basis. Furthermore, users can deploy SDN services on top of their VN topology by configuring the switch logic as required. Users do not have to deal with slicing restrictions, as the flowspace assignment is handled transparently by the hypervisor. As a consequence, each network slice can be regarded as a virtual SDN. In the remainder of this section, we discuss the requirements for SDN virtualization and present an overview of the proposed SDN virtualization layer.

A. Requirements for SDN Virtualization

We consider a substrate network composed of arbitrarily interconnected programmable switches. In particular, we assume an SDN hardware architecture compliant with OpenFlow [10], which is currently the only widely available SDN technology. As such, the switch forwarding tables can be programmed by external controllers using rule/action tuples, where one or more actions are performed on each incoming packet matching a set of header patterns specified in the rule. We purposely employ this generic model, as more advanced switch features may not be widely available in heterogeneous SDN environments. Furthermore, a virtual SDN is defined as a set of virtual nodes interconnected by virtual links. Each virtual node contains at least one flow table and a number of virtual interfaces attached to virtual links. Any virtual node essentially appears as a programmable (e.g., OpenFlow-enabled) switch to the vSDN user.

The ability to setup and operate vSDNs raises the following requirements which we address in this work:

**Flowspace Allocation.** In a virtualized SDN, each switch flow table can be modified by multiple vSDN users. It is therefore necessary to transparently allocate appropriate vSDN user flowspaces, so that user flow table access does not interfere with coexisting SDN slices.

**Arbitrary vSDN Topologies.** Depending on the vSDN embedding, vSDN links can span multiple hops. Consequently, the vSDN setup may require the installation of forwarding entries in intermediate switches, which are not part of the vSDN topology. In addition, it may be desirable to host multiple virtual nodes on a single switch if the number of virtual nodes exceeds the number of substrate switches or due to specific user requirements. Thereby, the flow table configuration for such a switch should correspond to the user-specified forwarding actions for all the virtual nodes assigned onto the switch.

**Live Migration.** Virtual nodes and links should support transparent migration across the SDN substrate in response to resource failures or substrate topology changes. To this end, the reconfiguration process, i.e., the installation of flows necessary for migration, should be executed in a well-defined, coordinated fashion in order to avoid traffic disruption.

Currently, FlowVisor [16] can be used to create SDN slices that constitute a subset of the substrate topology. The creation of arbitrary topologies, as discussed above, induces significant configuration overhead for the substrate operator. We aim to minimize this configuration effort by automating the switch flow table and flowspace configurations. This can relieve vSDN users and network operators of the burden of SDN configuration, allowing them to focus on the actual service deployment. To this end, we present a design for an SDN hypervisor that satisfies the aforementioned requirements.
B. SDN Hypervisor Overview

The SDN hypervisor constitutes a transparent layer between the SDN substrate and the vSDN operators (Fig. 1), facilitating the embedding, deployment and management of vSDNs. More precisely, the hypervisor performs the following tasks:

vSDN Setup Automation. The hypervisor generates and installs the switch flow table configurations required to setup a requested vSDN. A vSDN specification typically consists of a network topology and a set of attributes for nodes and links, such as switching capacity, flow table space, and link bandwidth. A prerequisite for vSDN setup is resource assignment which can be carried out using existing virtual network embedding algorithms (e.g., [6], [19]). To this end, the hypervisor maintains a global view of the substrate topology and resource availability, e.g., by periodically querying the substrate provider. Upon the completion of resource assignment, the hypervisor generates and installs the required flow table entries (e.g., forwarding, encapsulation) to all relevant switches. Furthermore, the hypervisor allocates the required flow table space in the switches where the virtual nodes are mapped. The required flowspace isolation among vSDN users is discussed in Section III-A. Similar to FlowVisor, bandwidth isolation is enforced by configuring traffic schedulers (e.g., weighted fair queueing) in the switch ports, since each vSDN can use a separate queue in every corresponding switch port.

Control Message Processing. The hypervisor restricts the view of a user to his own virtual network, which appears as an isolated SDN. Besides the vSDN topology, the hypervisor exposes a logical flow table to each vSDN user. Consequently, each vSDN user can use his own controllers to install the arbitrary flow table configurations. The hypervisor intercepts all user control messages, resolves the corresponding physical switches, ports, and queues, and translates the messages into the appropriate flow table entries, which are subsequently installed in the switches. Inversely, control messages generated by the switches (e.g., arrival of a new flow) are processed and forwarded only to the corresponding vSDN users, after they have been translated accordingly. The processing of control messages is completely transparent to the user giving him the illusion of exclusive flow table access. A prerequisite for control message rewriting is the translation between virtual link identifiers (encoded in packets) and flow table identifiers. We discuss the control message translation in Section III.

Live Migration. The hypervisor automates vSDN node and link migration by modifying the switch flow table configurations, as needed, i.e., the source and destination switch as well as all switches along the source and destination physical path for each virtual link under migration. In Section IV, we exemplify how the SDN hypervisor performs virtual link migration without any manual intervention and with minimal service disruption.

III. VIRTUAL SDN DEPLOYMENT

In this section, we explore the design space for SDN virtualization. We particularly discuss flowspace allocation, the encoding of virtual link identifiers, the translation between flow table and virtual link identifiers, and the deployment of arbitrary vSDN topologies.

A. Flow Table Identifiers

We consider an SDN switch contains a lookup table, populated remotely with flow entries by a single user. Each flow table row consists of a match rule and pointer to a set of actions which are applied whenever an incoming packet matches the rule bit pattern. Each table column corresponds to a matchable packet attribute such as a packet header field, or metadata like the input port. The cells of each row may contain either a specific match pattern or a special wildcard value. Multiple rows should not contain an identical set of attributes. Hence, using database terminology, the set of all table columns comprises a unique key for the table.

In a virtualized switch, the physical flow table is shared among multiple users. It is therefore possible that two user controllers will attempt to install identical match entries in their respective virtual flow tables. To avoid such conflicts and ensure consistency of the table, an additional column must be appended to the flow table which ensures that each “virtual” flow table hosted on the switch is mapped to an individual flowspace. In other words, we ensure that the physical flow table has a unique key. We refer to this new table attribute as the virtual flow table identifier (VTID), which must meet the following requirements. First, the VTID bit pattern should unambiguously associate a subset of the flow table entries with a specific user. Furthermore, the identifier should be unique at least at the switch level (it may also be globally unique). Second, the VTID should be independent of the remaining user-defined table attributes, i.e., an incoming packet should be able to match the VTID regardless of any additional bit patterns specified by the users. As a result, the VTID partitions the switch flow table into multiple virtual flow tables, which can be programmed by users without interfering with concurrent flow entries. Note that the VTID is managed by the hypervisor and is not exposed to the vSDN user.

In a virtualized switch, flow match rules are defined using virtual rather than physical interfaces. Hence, we extend the flow table with an identifier for virtual input ports (VPID). Users may specify the VPID directly or using a handle provided by the hypervisor. Each VPID has at least a virtual switch-level scope and is associated with a single virtual link.

It is important to highlight that the VPID cannot be used as a substitute for the VTID, even if each port is assigned a globally unique ID. This is because the uniqueness of all flow table entries cannot be guaranteed, as the column may contain multiple wildcard values. For example, two users may specify an identical rule in which the virtual input port is wildcarded, thus creating an ambiguity in the table.

B. Packet Identifier

In a virtualized network, packets must carry an additional identifier to ensure that all packets traversing the same physical link can be unambiguously mapped to a specific virtual link. It is mandatory that the so-called virtual link identifier (LID) is globally unique to ensure that packets can be forwarded between two virtual nodes along a path of intermediate physical
hops without interfering with concurrent virtual links. Note that unique LIDs can be generated by concatenating any single globally unique ID with multiple local IDs. Hence, the LID may carry additional information.

The hypervisor transparently installs the switch actions required to append link identifiers to each packet. We assume the use of some suitable tunneling mechanism such as MPLS/VLAN. The tags are updated at every virtual node. Depending on the size of the required ID space, the LID may be spread over multiple header fields or stored using only certain bits of a single header. In the latter case, the ID must be resolved using a bitmask in the switch flow table. OpenFlow supports actions for pushing/popping VLAN and MPLS tags, as well as arbitrary masking of VLAN tags since version 1.3.

C. Packet Identifier Resolution

Since the flow table lookup is performed using information encoded in the packet, we must ensure that the LID can be mapped to the VTID/VPID tuple used in the switch. A straightforward solution is to encode the packet and flow table identifiers using an identical format. As an example for this direct mapping, consider a tuple consisting of globally unique user ID, and virtual link ID with a user level scope. The tuple represents a globally unique LID and is therefore a valid packet identifier. Furthermore, the tuple elements also meet the requirements for a VTID and VPID respectively (as there is a 1:1 correspondence between a virtual port and an attached virtual link).

To generalize this mapping constraint, we introduce the notion of ID resolver which translates the LID to the VTID/VPID pair (see Fig. 2). A requirement for the ID resolver is that the LID is mappable to the VTID/VPID pair. In the case of direct mapping, the ID resolver is a null operator.

This concept enables a more flexible identifier encoding by using indirect mapping. To this end, an additional lookup step is introduced, which translates the LID into a format compatible to the two flow table identifiers. For example, consider a globally unique LID. Given an additional resolver table, the LID can be mapped directly to a locally unique VTID and VPID using an initial lookup. The identifiers can then be used to perform a second lookup in the standard flow table. A drawback of this approach is that multiple lookups might not be supported by the switch hardware. On the other hand, the remapping of LIDs is significantly simplified, as only a single resolver table row must be updated rather than all virtual flow table entries associated with the link. Note that compared to

[1] It is possible to reuse LIDs in non-overlapping network segments. In this case, LID assignment is similar to the NP-hard routing and wavelength assignment (RWA) problem. Hence, this approach significantly complicates the deployment of vSDNs.

Fig. 2. The ID resolver ensures that the link identifier can be mapped to a virtual flow table.

Fig. 3. Packet forwarding example using a resolver table stored on the switch flow table along with user-specified flow entries.

The number of regular user flow entries, the size of the resolver table is negligible.

One option for implementing the resolver mapping is to utilize additional tables exposed by the switch hardware (e.g., through OpenFlow). The resolved IDs may overwrite the packet header containing the LID or use additional metadata fields allocated internally by the switch. Alternately, in switches which do not expose additional tables, the resolver table can be stored alongside the regular user flow table entries, as long as the flowspace of the two tables do not overlap. In this case, after the resolver lookup, packets are redirected to the ingress of the switch pipeline using a special logical loopback interface (P_Lo), as depicted in Fig. 3. Such logical interfaces are supported in the current OpenFlow specification, however, their implementation is optional and vendor specific. We will show in the following sections that such a feature can simplify the implementation of some advanced SDN virtualization features.

D. Deploying Arbitrary vSDN Topologies

We now discuss the support for arbitrary vSDN topologies and particularly the deployment and configuration of multiple interconnected virtual nodes on top of a single physical switch. This requires that multiple virtual flow tables (and potentially the corresponding resolver tables) are stored in the same substrate switch flow table. Since the output port of each flow table entry is associated with a unique LID, any vSDN topology can be embedded onto a single switch using a

Fig. 4. Multiple interconnected virtual nodes mapped onto a single switch using a loopback interface.
loopback interface which reinserts packets to the ingress of the switch pipeline. As a result, any packet forwarded through a set of virtual nodes will traverse the flow table multiple times before reaching a physical switch output port, as depicted in Fig. 4. To avoid infinite loops caused by user misconfiguration, the loopback port can assign a TTL meta field to each packet to limit the number of iterations a packet is allowed to perform.

Deploying multiple virtual nodes on top of a switch without a loopback interface is more challenging, since there is no simple way to combine multiple flow tables and the corresponding output actions. In this case, we outline another technique in which the hypervisor analyzes the flow tables of the virtual topology and constructs a single flow table which is functionally identical to the virtual topology configured on the switch. To that end, for every external virtual link the hypervisor inspects each flow entry in the corresponding table and the associated actions, and generates a pseudo header containing the resulting packet modifications. The output port of the entry is used as a pointer to the next virtual flow table, where the pseudo header is used to perform a subsequent lookup. The process is repeated till a flow entry associated with a physical switch output port is reached. The pseudo header and the actions of the egress flow entry are used to construct a flow entry which is logically equivalent to the chain of flow entries. The computation of such a table is resource-intensive. Furthermore, complex mechanisms are required to deal with circular rules and flooding actions. Therefore, the applicability of this technique is limited to a small number of virtual nodes per switch.

IV. VIRTUAL SDN RESOURCE MIGRATION

Live migration enables network operators to seamlessly reassign VN topologies to different substrate resources with minimal or no traffic disruption. In this section, we discuss two techniques which can be used by the SDN hypervisor to implement a seamless migration in vSDN topologies without manual reconfiguration. We decompose vSDN migration into two operations: (i) virtual node migration, i.e., the transfer of the virtual node flow tables to the destination switch, and (ii) virtual link migration, i.e., traffic redirection from the source to destination path. To avoid traffic disruption, the hypervisor must ensure that the appropriate flow entries are installed at the link endpoints, and along each physical hop of the virtual link throughout the duration of the migration process. Recall that each virtual link is associated with a globally unique identifier encoded within each packet and each intermediate hop contains a corresponding forwarding entry.

Virtual link migration using a new link identifier. This technique relies on the allocation of a new link identifier \( L' \) to the virtual link. First, new forwarding entries using \( L' \) are installed along each hop of the link path \( p' \). Subsequently, the hypervisor prepares the endpoint hosts A and B for receiving packets over the new virtual link. To this end, all flow table entries containing the original link ID \( L \) must be temporarily duplicated. Finally, the hypervisor redirects traffic over the new path by exchanging all flow actions containing a reference to \( L \) with \( L' \), and updating the corresponding physical ports. Note that the rewriting operation does not have to be synchronized, since traffic can still be forwarded via the initial path as long as the migration is in progress. Once all flow table entries have been updated, the flow entries associated with the original link ID \( L \) and the corresponding intermediate hop forwarding entries can be removed. The original link ID is now free for reuse.

A drawback of this technique is that temporary flow entries must be maintained on both virtual link end-points. This can result in excessive usage of flow table space, since all user flow table entries that include link IDs must be cloned. The use of resolver tables, as discussed in Section III-C, can mitigate this. More precisely, if the link IDs mapping is stored in a resolver table, each virtual link ID corresponds to a single resolver table entry and consequently, only that entry needs to be cloned.

Asynchronous virtual link migration. Alternatively, virtual links can be migrated without modifying LIDs. To this end, the hypervisor references each virtual link using two identifiers: for the forward and reverse directions, respectively. Consequently, each directed virtual link can be migrated asynchronously. The key aspect of this migration technique is that the forwarding entries for each hop along the path are installed by the hypervisor in the direction opposite to the traffic flow. As a result, a valid forwarding path is maintained throughout the migration process and no traffic is dropped.

To illustrate this migration technique, we consider the scenario in Fig. 5, where a virtual link currently mapped to path \( p \) with LID=101 is migrated to a new “destination” path \( p' \). The traffic direction is from node A to node B and \( i_2, i_3 \) are intermediate nodes. When the migration is initiated, a forwarding entry is first installed at intermediate node \( i_5 \). Subsequently, corresponding rules are installed at \( i_1, i_3, \ldots, i_1 \). Note that the rule at (3) overwrites the previous flow table entry, and therefore traffic temporarily flows over the path \( i_2, i_3, i_4, i_5 \). In the last step (6), all node A entries associated with LID=101 are updated to point to the new physical port \( p_2 \).

It is important to highlight that while both techniques prevent packet loss during migration, packet reordering may occur in cases where the destination path is longer than the original path.

Virtual node migration. Virtual node migration requires the transfer of all corresponding flow table entries and the redirection of all attached virtual links to the destination switch. To
avoid traffic disruption, the hypervisor first clones the virtual node flow table and maintains two copies. Subsequently, the links are migrated using one of the mechanisms discussed above. After the migration of all virtual links, the original flow table and intermediate forwarding entries are removed.

V. SCALING THE SDN HYPERVISOR

In the following, we discuss techniques enabling scalability for SDN hypervisor and thereby allowing the concurrent deployment of a large number of vSDNs, while handling vSDN configurations from multiple tenants. We discuss two main scalability issues of the control and the data planes. As all user control plane traffic must be processed by the SDN hypervisor, we use a distributed hypervisor design to ensure that sufficient processing resources can be provisioned. In the data plane, we address the problem of limited memory of the switches, by offloading some of the forwarding decisions to dedicated servers.

A. Control Plane Scalability

The distributed SDN hypervisor, depicted in Fig. 6, consists of a management module (MM) and multiple controller proxies (CPX) used to evenly distribute the control load. The substrate SDN is subdivided into multiple SDN domains, to each of which a dedicated CPX is being assigned. CPXs are responsible for operations within the corresponding domains, while the MM performs SDN-wide operations, such as vSDN topology mappings, SDN domain segmentation, and network-wide migrations. The proposed functions of the MM and CPX are summarized in Table I. We assume that all SDN domains are operated by a single SDN provider, which implies that a CPX can have visibility to other SDN domains if needed. SDN virtualization across administratively independent SDN domains is more challenging due to the policies of network operators that restrict information disclosure and interoperability with other parties [7]. Multi-provider SDN virtualization can be tackled by leveraging on existing network virtualization architectures [15], [20] and VN embedding frameworks [7].

The distributed SDN hypervisor transparently performs the deployment and operation of vSDNs, as follows:

vSDN Deployment. Upon receiving a vSDN embedding request, the MM initially computes the mapping of the vSDN topology to the substrate network. Given the large body of work on VN embedding (e.g., [6], [7], [19], [9]), vSDN mapping is deemed tractable and can be achieved by introducing additional constraints (e.g., available flowspace in switches) to VN embedding formulations or heuristic algorithms. Subsequently, the MM assigns the resources selected for the requested vSDN to the CPXs, according to the SDN domain segmentation. In addition, the MM assigns to the CPXs unique identifiers, as discussed in Section III.

vSDN setup is further carried out with the installation of the required flow entries in the switches. To this end, each CPX deploys the allocated topology segment by installing infrastructure flow entries, which unambiguously bind traffic to a specific logical context using tagging.

vSDN Operation. After the vSDN has been deployed, the SDN hypervisor facilitates its operation and configuration. Since isolation between tenants is essential, each CPX performs policy control on the flow table accesses and ensures that the resulting flow entries are mapped onto non-overlapping flowspace. All control communication between a tenant’s controller and the forwarding plane is redirected through the CPX responsible for the corresponding switch. Before installing a tenant flow entry to a switch, the CPX rewrites the control message, such that all references to virtual resources are replaced by the corresponding physical entities, and appropriate traffic tagging actions are appended. The state of each virtual node in a given SDN domain is maintained solely by the corresponding CPX. Consequently, each CPX can independently migrate vSDN components (i.e., switches or links) in order to optimize resource utilization within the domain. Global optimizations are coordinated by the MM.

B. Data Plane Scalability

Data plane scalability limitations primarily stem from the relatively small flow table size of OpenFlow switches, which
is typically in the order of several thousands of entries. This essentially restricts the number of vSDNs that can be concurrently deployed and essentially reduces the revenue that a vSDN provider can generate. To mitigate this problem and incentivize network providers to lease vSDNs to tenants, we deploy so-called auxiliary software datapaths (ASD) in the substrate network, similarly to [18]. OpenFlow-enabled ASDs (e.g., OpenVSwitch [11]) are hosted in commodity servers deployed in each SDN domain. Servers are equipped with a large amount of memory, sufficient for storing full copies of logical flow tables. Despite the recent advances in commodity hardware (e.g., non-uniform memory architectures, network cards with hardware multi-queueing) and technologies for high-performance packet I/O (e.g., netmap [12]), the divide between commodity and specialized hardware still remains, with the latter offering at least one order of magnitude larger switching capacity.

To address this issue, we exploit the Zipf property of aggregate traffic, i.e., the fact that a small fraction of flows accounts for most of the traffic volume. More precisely, the small number of flows that carry most of the traffic are handled by the OpenFlow switches, while the remaining (mice) traffic is handled by the ASDs. To this end, a set of low-priority infrastructure entries redirect traffic from the domain edge to the ASD, when no high-priority flow entry has been cached at a domain switch (i.e., acting as a fallback path) [2]. Each CPX selects the flow entries that will be cached, so that most of the traffic is offloaded from the ASD. In addition, the CPX ensures that cached flow entries do not alter the semantic integrity of the flow table rules by re-encoding flow entries as needed. Traffic offloading can be achieved using traditional caching techniques (e.g., LFU, LRU), or a recent technique which yields higher efficiency [14]. Since these techniques have been evaluated for IP forwarding, their efficiency in SDN virtualization may be limited, due to the diversity of the forwarding entries installed by the tenants.

VI. RELATED WORK

In this section, we briefly discuss systems related to the proposed SDN hypervisor. FlowVisor [16] enforces flow table isolation by filtering and rewriting control messages, so that each user can view and control only a subset of flow entries. However, FlowVisor provides support for SDN slicing rather than SDN virtualization. As such, virtual SDNs deployed using FlowVisor may only utilize a subset of the substrate resources, rather than arbitrary topologies. Further, SDN slice deployment and operation using FlowVisor induces significant configuration and planning overhead for SDN operators. In contrast, our SDN hypervisor extends FlowVisor to orchestrate the deployment and operation of vSDNs, obviating the need for manual configurations, by generating vSDN topology mappings, as well as the flowspaces and forwarding entries required for vSDN setup and resource migration.

A high-level design of a network hypervisor for virtualizing SDN substrates is presented in [4]. In addition the authors present a number of motivating use cases for a virtualized control plane. Our work expands on a number of key aspects necessary for the implementation of a transparent network hypervisor, such as mapping between virtual and physical resources, flow space isolation, as well as migration mechanisms.

ADVisor [13] provides a layer for the creation of basic vSDN topologies. However, it lacks support for vSDN topologies, where multiple virtual nodes may need to be deployed on top of a single switch. Our work enables the deployment of fully arbitrary topologies and carries out vSDN resource migration without any manual SDN reconfiguration.

Authors in [5] outline the architecture of a distributed SDN controller, namely Onix, with emphasis on control plane scalability. Our scalable control plane architecture extends this work for SDN virtualization orchestration. We further address the switch flow table size limitations to scale the SDN forwarding plane.

Furthermore, our proposed techniques for arbitrary vSDN topology creation and vSDN migration can be applied to the upcoming OpenVirtex platform [3], enhancing its versatility in terms of SDN virtualization.

VII. CONCLUSIONS

In this paper, we presented a SDN hypervisor that orchestrates the deployment and operation of virtual SDNs on top of a shared substrate. In this respect, we bring the benefits of SDNs into the network virtualization domain, allowing the tenants to lease virtual networks which can be configured and controlled as SDNs. Once a vSDN has been deployed, the tenant is able to use a controller and install flow entries to each virtual switch flow table, as if accessing a dedicated physical SDN. Therefore, tenants will be able to take control of the virtual networks granted to them, implementing custom forwarding decisions and security policies, while configuring access control as needed. At the same time, the hypervisor relieves SDN operators of the configuration overhead during vSDN topology mapping, deployment, and resource migration.

We investigated techniques for selecting, encoding and mapping identifiers for link virtualization and flowspace allocation, and discussed their implementation with the recent OpenFlow specifications. We also exemplified techniques for the deployment of arbitrary vSDN topologies, and for seamless vSDN node and link migration. Furthermore, we elaborated on techniques for the scalability of the hypervisor. To this end, we delegate the operations on each SDN domain to a separate controller proxy, while a management module is responsible for the vSDN topology mapping and network-wide operations. We deploy OpenFlow-enabled ASDs in each domain to mitigate the limited flow table size of switches. SDN data plane scalability also raises the need for efficient traffic offloading mechanisms, given the diversity of forwarding rules that can be installed by the tenants to configure routing, security policies, or access control.

The distributed SDN hypervisor architecture certainly entails more challenges, which we are currently investigating using a prototype implementation consisting of Pronto 3295 switches and commodity servers hosting ASDs (implemented with OpenVSwitch) and hypervisor modules (i.e., MM and CPXs), implemented with FlowVisor and NOX. We are particularly studying the interplay between vSDN topology embedding and SDN segmentation in order to maintain load balancing as vSDNs are being allocated or released, and we are investigating the collaboration among CPXs for network-wide resource optimizations.
ACKNOWLEDGMENTS

This work was partially supported by the EU FP7 T-NOVA Project (619520).

REFERENCES


