UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing

Yanfang Le
University of Wisconsin-Madison
yanfang@cs.wisc.edu

Hyunseok Chang
Nokia Bell Labs
hyunseok.chang@nokia-bell-labs.com

Sarit Mukherjee
Nokia Bell Labs
sarit.mukherjee@nokia-bell-labs.com

Limin Wang
Nokia Bell Labs
limin.1.wang@nokia-bell-labs.com

Aditya Akella
University of Wisconsin-Madison
akella@cs.wisc.edu

Michael M. Swift
University of Wisconsin-Madison
swift@cs.wisc.edu

T.V. Lakshman
Nokia Bell Labs
t.v.lakshman@nokia-bell-labs.com

ABSTRACT

Increasingly, smart Network Interface Cards (sNICs) are being used in data centers to offload networking functions (NFs) from host processors thereby making these processors available for tenant applications. Modern sNICs have fully programmable, energy-efficient multi-core processors on which many packet processing functions, including a full-blown programmable switch, can run. However, having multiple switch instances deployed across the host hypervisor and the attached sNICs makes controlling them difficult and data plane operations more complex.

This paper proposes a generalized SDN-controlled NF offload architecture called UNO. It can transparently offload dynamically selected host processors' packet processing functions to sNICs by using multiple switches in the host while keeping the data center-wide network control and management planes unmodified. UNO exposes a single virtual control plane to the SDN controller and hides dynamic NF offload behind a unified virtual management plane. This enables UNO to make optimal use of host's and sNIC's combined packet processing capabilities with local optimization based on locally observed traffic patterns and resource consumption, and without central controller involvement. Experimental results based on a real UNO prototype in realistic scenarios show promising results: it can save processing worth up to 8 CPU cores, reduce power usage by up to 2x, and reduce the control plane overhead by more than 50%.

CCS CONCEPTS

• **Networks → Network components**; Middle boxes / network appliances;

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3132252

KEYWORDS

Networking and SDNs, Virtualization and containers

ACM Reference Format:
Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T.V. Lakshman. 2017. UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 14 pages.
https://doi.org/10.1145/3127479.3132252

1 INTRODUCTION

Modern software defined networking (SDN)-based data centers are architecturally "edge-based" [36, 41, 84], where a variety of infrastructure components are housed at end-hosts (referred to as hosts, henceforth). That is, in addition to tenant virtual machines (VMs), data center providers run virtual instances of a variety of network functions (NFs), (e.g., firewalls, NATs, load balancers, etc.) at hosts. Each host also runs a software switch such as Open vSwitch (OVS) [81] to handle network communications amongst the tenant VMs, NFs, and any remote entities. Because it is crucial to ensure data center compute is maximally used toward tenant applications, data center providers often strive to minimize the compute resource consumption of their infrastructure components, i.e., NFs and software switches.

Unfortunately, this has become increasingly difficult in recent years due to a confluence of key technology trends. First, the speed of data center interconnects continues to increase [3], as a result of which NFs must now process many more packets per second, at significant compute cost. Second, as more and more VMs and lightweight containers are provisioned on a host, the switching load on software switches also increases so as to support a large number of virtual ports [57]. Finally, as virtual networking at the edge becomes more sophisticated, new types of packet processing functions are being deployed [36, 60, 78, 80, 96], and this also translates into increasingly more CPU cycles to execute complex logic for the same amount of traffic at the host. These factors are causing increasingly large fraction of host processors to be dedicated to packet processing operations in NFs and software switches, leaving a smaller fraction of host CPU free for running tenant workloads.

Smart Network Interface Cards (sNICs) are promising to offer some respite from these challenges. These sNICs offer energy-efficient processors [2, 12] that can be programmed to dynamically offload custom-built packet processing functions from the host to the sNIC, which offsets the increasing infrastructure component cost at the edge.

However, leveraging sNICs effectively is challenging due to three reasons. First, sNICs have limited total compute and memory capabilities, so not all NFs and switching can be hosted on the sNICs. Second, hosting switching entirely on the sNIC, while feasible, imposes high latency costs (for packets that traverse between NFs deployed off the sNIC and on the host; see Fig. 2) and imposes unnecessary bandwidth overhead due to multiple traversals across the host PCI Express (PCIe) bus. Third, splitting switching and the set of NFs across the host hypervisor and the sNIC leads to a significant SDN management burden. The data center-wide SDN controller must manage twice as many switches (two switches per host – one each in the hypervisor and the sNIC) and also handle the much more difficult task of managing potential migration of NFs between the host and sNIC based on evolving traffic load patterns. We elaborate on these issues in Section 2.3.

To address these challenges and make it easy to leverage sNICs for switching and NFs in modern virtualized data centers, this paper proposes UNO, a generalized SDN-controlled NF offload architecture. UNO splits switching between the host software and the sNIC. It includes an NF agent, which abstracts the existence of the sNIC away from the SDN controller. UNO dynamically uses sNIC resources to augment the host's packet processing capabilities by offloading subsets of switching and NFs to the sNIC. In particular, UNO uses a novel linear programming formulation to determine the optimal placement for an NF (at the host software or on the sNIC) based on currently-observed traffic patterns and the load the NF's processing entails. UNO also includes OneSwitch, which translates rules from the data center-wide SDN controller for NF traversal (i.e., the specific chain of NFs that traffic must go through, as dictated by the tenant) into rules at the local host and sNIC switches. We develop a novel rule translation algorithm to this end.

The offload decisions in UNO remain with the host to make it timely, and also to improve efficiency and scalability compared to making all decisions at the SDN controller. UNO also provides a uniform single-switch management interface for each host, whether or not the host is equipped with an sNIC, which simplifies data center-wide network management. UNO achieves all this without requiring any changes to the data center's centralized management and control planes.

We have prototyped UNO and conduct detailed experiments to show its efficacy. We find that UNO can potentially save processing worth up to 8 host CPU cores in a SD-WAN setting [26], and can reduce power by 2x. We study the corresponding costs in migrating NFs, along with the dynamic internal state that NFs maintain across the PCIe bus. We show that UNO can optimally leverage sNIC's full resource capacity in the presence of dynamic traffic changes by via dynamic offloading. Compared to exposing the sNIC entirely to the SDN controller, we demonstrate that UNO can reduce the control plane overhead (e.g., number of flow rules maintained by the controller) by more than 50% by concealing the dynamic offload decisions.

In summary, the key contributions of our paper are as follows: (i) We present a generalized SDN-controlled offload architecture for dynamically making the best use of sNICs and host packet processing capabilities, without requiring any changes to the management and orchestration of the data center. (ii) We design and implement a proof-of-concept system for this architecture. (iii) We provide a rule translation algorithm that can map NF traversal rules from an external controller and to the constituent host/sNIC switches so that packet routing semantics are correctly enforced. (iv) We formulate and implement an NF placement algorithm that dynamically selects the best location to place an NF so that the host, sNIC and interconnection resources are utilized optimally. (v) We implement an NF migration procedure that can relocate an NF, together with its dynamic internal state, between the host and the sNIC at runtime such that packet losses are eliminated and internal state remains up-to-date even as it is being relocated.

2 BACKGROUND AND MOTIVATION

2.1 Network Function Virtualization

Network Function Virtualization (NFV) [54] is a key technology trend that enables data center operators to realize various NFs (e.g., firewall, load balancer, IDS) as virtual appliances running on top of commodity server hardware, replacing dedicated hardware appliances. In a cloud environment where heterogeneous tenant applications co-exist with dynamically changing requirements, the role of NFs is critical to ensure the cloud infrastructure's performance/cost scalability, resilience, security protection, fast innovation, etc. Operating virtual NFs themselves requires the data center's shared infrastructure resources, and therefore NFs are created and managed by the Infrastructure Manager (e.g., OpenStack [19]), under the control of the centralized NFV orchestrator [18, 21]. Within the Infrastructure Manager, the NF controller deploys NFs on end servers, and the SDN controller programs the data center network using SDN rules to steer traffic through deployed NFs as per NF traversals – or "NF chaining" – specified, e.g., by tenants.

2.2 Network Interface Cards

Traditional NICs [9, 13] provide several pre-packaged functions to offload routine packet processing to the NIC (e.g., checksum computation, transmit/receive large segment offload, tunnel offload, flow hashing and interrupt coalescence). They do not provide any programmability to create a new packet processing function or to chain the functions selectively on certain flows.

Advanced NICs equipped with special hardware (ASIC) are purpose-built to offload pre-defined packet processing functions (e.g., OVS fastpath [14, 17], packet and flow filtering [1]). These functions are more advanced than what traditional NICs support. Some NICs can also be enhanced with FPGAs for custom function development, and can be used to offload host functions. Examples include offloading distributed consensus protocols [45, 59, 69], memcached and key-value stores [38, 70, 94], rate-limiting packet flows [83], cryptography, quality of service and storage networking [49]. We refer to these NICs as *hardware acceleration NICs*. They are application specific, and offer none or hard-to-program customization [46] of new functions. High-level programmable platforms like P4 [39] support limited network functions. Also, existing hardware offers

no systematic methodology for chaining multiple offload capabilities, nor does it have the capability to implement complex network functions, such as encryption or deep packet inspection.

In our context, *smart NIC (sNIC)* refers to a NIC equipped with fully-programmable, system-on-chip multi-core processor on which a full-fledged operating system can execute any *arbitrary* packet processing functions [2, 12, 17, 28]. With a much higher level of flexibility and programmability, these sNICs can offload almost any packet processing function [75, 88] from the host hypervisor, thereby offsetting the increasing infrastructure components' cost at the edge. In addition, they utilize much more energy-efficient processors (compared to x86 host processors), achieving higher energy efficiency in packet processing [53]. In particular, for server-based edge networking in virtualized data centers, where workloads can change frequently, these sNICs if used carefully can lead to substantial improvements in host CPU utilization and power consumption. As such, they are increasingly being deployed in data centers to address the deficiencies of traditional and hardware acceleration NICs [4, 25, 29].

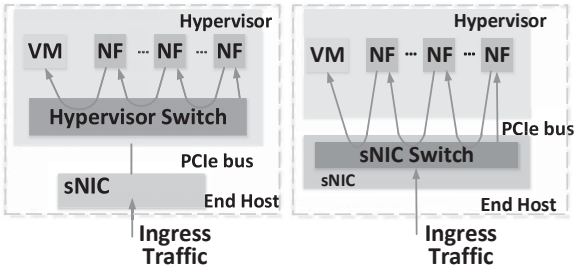
2.3 Motivation

Clearly, sNICs can offer wide flexibility in developing NFs for both host and sNIC platforms, and running the NFs on either platform on demand. However, there is no general framework to easily and intelligently offload and/or service chain the functions across hosts and sNICs. Programming frameworks through which application developers can specify the scope of offload based on (smart) NIC's capabilities have been studied [35, 76, 93, 98]. These models may require rewriting NFs to conform to the prescribed API, and they may not support tenant-specific customization and runtime dynamism required in a data center. We provide this flexibility by leveraging SDN control on sNICs to dynamically steer the traffic flow to a network function (running on either platform).

In this context, a straightforward approach is to offload switching entirely to the sNIC [17, 71] and steer all packet flow from there. VMs and NFs running on the host then bypass the host hypervisor (e.g., via SR-IOV) and connect to the offloaded switch directly [58]. Such full switching offload keeps the control plane unmodified, but it introduces overhead in the data plane.

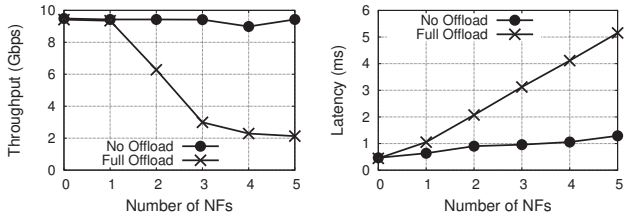
For example, when traffic flows across VM/NF instances within the same host [58, 97], which is increasingly common due to service-chained NFs, microservices-based applications [47], and zero trust networking [30, 43, 67, 77], the intra-host flows must cross the host PCIe bus multiple times back and forth between the hypervisor and the sNIC [86]. This restricts the local VM-to-VM and VM-to-NF throughput as memory bandwidth is higher than PCIe bandwidth [68], but equally importantly it negatively affects per-packet latency.

To validate these points, we run the following illustrative experiment using the setup shown in Fig. 1. In one case, we chain VM/NF instances using the hypervisor switch ("no-offload"), while in the other case, the entire switching is offloaded to the sNIC via SR-IOV ("full-offload"). We compare these cases by measuring the throughput and latency of the chain while varying the number of chained NFs. Fig. 2(a) shows that in the no-offload case, the throughput is not much affected by the number of chained NFs, while fully



(a) No Offload. (b) Full offload.

Figure 1: NF chaining experiment setup.



(a) Throughput. (b) Latency.

Figure 2: No offload vs. full offload.

offloaded switching experiences significantly degraded throughput with more chained NFs. The latter is due to the sNIC's maximum PCIe bandwidth limitation. For example, with two chained NFs, throughput goes down to 6Gbps, due to 32Gbps maximum bandwidth supported by PCIe Gen2 x8 NIC card. Note that with a longer chain, not only the throughput of the chain, but also the *aggregate* PCIe throughput degrades significantly below the maximum PCIe bandwidth. With three chained NFs, for example, the aggregate throughput is only 21Gbps (3Gbps×7). This implies that while maximum supported PCIe bandwidth may be high, NICs may not leverage all available bandwidth due to CPUs and PCIe bus contention [87], which indicates that we should limit PCIe read/write. Fig. 2(b) shows a similar performance gap between the two cases, but in terms of latency. These results demonstrate that full switch offload may not be desirable for both throughput and latency reasons. Note that using kernel-bypass techniques can improve latency of offload, but PCIe overheads remain. While faster PCIe buses may reduce this contention, NF selection is still important given limited sNIC processing capacity and the added latency of crossing the PCIe bus multiple times for a single packet. Furthermore, faster line rates may again increase pressure on I/O bus bandwidth.

Performance aside, fully offloaded switching relies on hardware resources (SR-IOV virtual functions) to scale the number of ports. This is intrinsically more restrictive than software ports, especially considering emerging lightweight containers with ultra high deployment density [8], and the high port density supported by modern software switches (e.g., 64K ports for OVS).

Strawman solution: The inefficiency in intra-host communication is best addressed by using the hypervisor switch only and not offloading to sNICs. On the other hand, to flexibly use sNIC's

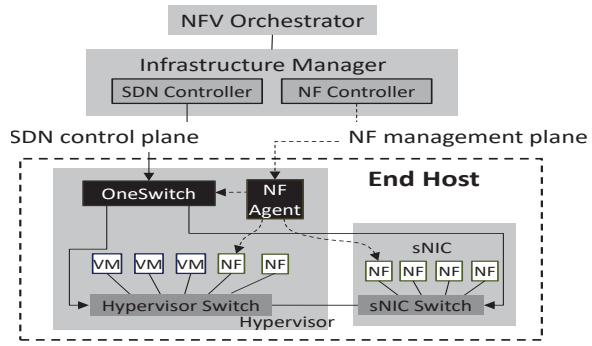


Figure 3: Host with UNO in an SDN-managed network. UNO components, OneSwitch and NF agent, are shown in black.

capabilities, we need to extend SDN control to the sNIC. Therefore, in addition to the hypervisor switch, we can operate an SDN-controlled switch at the sNIC (referred to as an sNIC switch[1]), to enable selective and flexible offload of NF packet processing and to extend service chaining to the sNIC.

The hypervisor switch connects to all tenant VMs and NF instances running on the hypervisor, while the sNIC switch connects to offloaded NF instances. The two switches are logically interconnected via a virtual port pair over the PCIe bus. Then, depending on where NFs are deployed with respect to associated tenant applications, either switch can be used by the SDN controller to set up a complete service chain.

This architecture, however, introduces additional complexity in the existing management/control planes as the data center controller now must control *more than one switch* per host. With even one sNIC per host, the number of switches, as well as the number of flows rules for chaining, that the controller would need to manage across the entire data center is doubled. Furthermore, the controller would need to decide which switch – hypervisor or sNIC switch – to connect NFs to and when to migrate NFs between the two switches (if necessary), and provision the switches accordingly. Placement is important to decide how to optimally use limited sNIC compute and memory resources towards NFs while offering optimal benefits at low cost. Migration is important because evolving traffic patterns may impose different load on different NFs over time, and may also impact the amount of data exchanged across pairs of NFs in a chain; it may therefore be necessary to re-place NFs across the host and the sNIC to re-optimize resource use, cost, and performance.

These activities require fine-grained resource monitoring and controlling of individual hosts, as well as making placement/migration decisions for the increasing number of NFs deployed data center wide. Frequent execution of this can severely limit the scalability of the management/control planes, while infrequent execution can limit the performance of the data plane. Moreover, if all hosts are not equipped with sNICs, or use different sNICs, the heterogeneity would bring in more management complexity into the control plane.

[1]The sNIC switch can be software-based or hardware-based depending on sNIC's capability.

We therefore seek a design that preserves the benefit of flexible NF placement across both the host and the sNIC, but minimizes the complexity exposed to the data center controller.

3 UNO ARCHITECTURE

UNO is a framework that systematically and dynamically selects the best combination of host and sNIC processing for NFs using local state information and without requiring central controller intervention. The goal of UNO is to selectively offload NFs to sNICs (if available) without introducing any additional complexity in the existing NF management and control planes.

3.1 Design Overview

UNO co-exists with a centralized NFV platform [18, 21] which deploys and manages NF instances for tenants on end hosts. Fig. 3 shows how UNO (represented in a dotted box) fits within the existing NFV platform. Note that UNO continues to leverage a logically central control plane spanning the entire infrastructure, reflecting typical SDN architectures. The key difference is that UNO's control plane is decomposed into per-host controllers in addition to a logically central entity. We describe design details below and argue that this way of structuring the control plane improves scalability and efficiency. Note that the NFV orchestrator and infrastructure manager (e.g., OpenStack) largely remain unchanged and agnostic to our end-host architecture.

UNO is a framework running on virtualized platforms that conceals the complexity of having multiple switches from a data center wide SDN controller and NFV orchestrator. Across both a hypervisor switch on the host and one or more sNIC switches, UNO manages (i) the placement of NFs and (ii) the enforcement of SDN rules. The SDN controller and NFV orchestrator are presented with the abstraction of a single virtual switch. Crucially, the details of where data flows and where NFs execute is handled by UNO, which reacts dynamically to traffic patterns, SDN rules, and the set of NFs installed. Delegating these issues to individual hosts offers better scalability than a single central controller. Hosts, where all the packet processing and tenant applications are running, are better suited to make optimal offload decisions based on local context (e.g., current host/sNIC resource utilization) than a remote controller.

UNO is split into two components in each end host: *network function agent* and *OneSwitch*, which are used for the management plane and the control plane, respectively. In the following, we describe them in more detail. UNO abstracts the sNIC away from the controller, which in effect keeps the host's interface to the controller unmodified.

UNO maintains *virtualized* management and control planes within the host. The virtual management plane abstracts out where (e.g., hypervisor or sNIC) NF instances are deployed, while the virtual control plane hides multiple switches on the host from the external controller. As shown in Fig. 4, the virtual control plane intelligently maps the hypervisor and sNIC switches into a single virtual data plane which is exposed to the SDN controller for management. When the controller adds a new NF instance or installs a flow rule into the virtual data plane, the NF instance is deployed on either switch by the local management plane decision, and the rule is

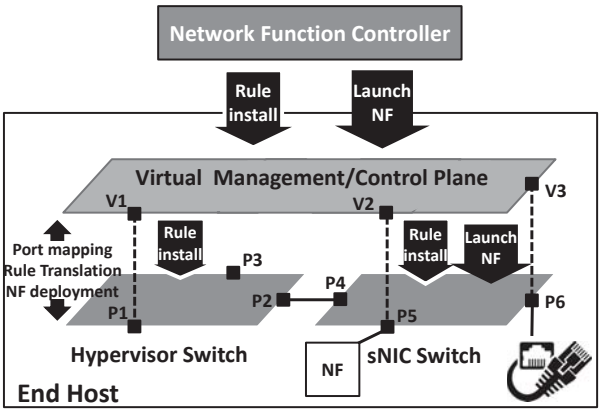


Figure 4: Virtual management/control plane.

mapped appropriately to the switches by corresponding control plane translation.

UNO addresses the following main challenges. First, when a new NF is deployed, the virtual management plane must decide on a placement for the function (hypervisor or sNIC) taking into consideration constraints such as current resource availability at the host and sNICs, the intra-host communication capacity, etc., while also minimizing host CPU usage. We develop an optimal placement algorithm to address this (Section 3.2.1).

Once the NF placement decision is made, we need a rule mapper that can translate the rules sent by the controller to an equivalent set of local rules that can be instantiated at the local hypervisor and sNIC switches. The rule translation must handle rules that contain metadata (e.g., ports), and must carry the metadata across the switches to maintain correctness across the virtual-physical boundary. Our approach to address this issue is described in Section 3.3.1; it builds on recent advances in control plane virtualization [34, 40, 92].

To work in a dynamic environment, the allocated workload at the host and the sNIC must be refined periodically, e.g., with changes in traffic volume and compute load (e.g., when a new VM/NF joins/leaves the host). We propose a novel approach for runtime selection of candidate NFs to migrate between the hypervisor and the sNIC, followed by triggering necessary remapping for associated switch ports and rules in the virtual control plane (Section 3.2.2).

3.2 Network Function Agent

UNO's Network Function agent ("NF agent") makes the management plane agnostic to where (at the hypervisor or sNIC) NF instances are deployed. It is responsible for launching VM/NF instances and configuring OneSwitch (e.g., creating ports) according to management plane policies. On the host side, it incorprates additional intelligence to decide (without the NFV orchestrator's involvement) whether to deploy NF instances, on a hypervisor or sNIC (Section 3.2.1). Once the NF agent deploys an NF instance on the hypervisor or at the sNIC, it creates a new physical port on the corresponding host/sNIC switch, and connects the NF instance at the port. Finally the NF agent maps the physical port to an externally visible virtual port maintained by OneSwitch (Section 3.3).

3.2.1 NF placement decision problem. The decision on where to deploy an NF instance on a given host is driven by three criteria: (1) the hypervisor's/sNIC's current resource utilization, (2) current PCIe bandwidth utilization, and (3) sNIC's available hardware acceleration capabilities. The goal of NF placement decision is to offload as much NF processing workload to the sNIC as possible to free up hypervisor resources. sNIC offload is particularly beneficial if the offloaded NF can leverage hardware acceleration capabilities on the sNIC, as that can significantly reduce general-purpose core usage at the sNIC. The constraints for NF offload are: (1) aggregated offloaded workload on sNIC cannot exceed the sNIC's resource capacity, and (2) cross traffic over PCIe bus cannot exceed PCIe bandwidth limitation.

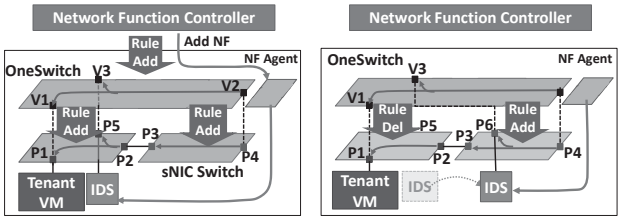
The NF placement problem is related to the classical *s-t* graph cut problem [11] which finds the optimal partitioning $C = (S, T)$ of vertices in a graph, such that certain properties (e.g., total edge weights on the cut) are minimized or maximized. In our problem, each NF/VM instance (deployed or to be deployed) is modeled as a vertex in a graph. sNIC's Ethernet ports are also represented as vertices. If there is direct traffic exchange between any two vertices, an edge is added to the graph with average throughput of the traffic as edge weight. We call this a *placement graph*. This maps the NF placement decision into a graph cut problem that finds $C = (H, N)$, where H is the set of NFs/VMs deployed in the host hypervisor, and N is the set of NFs or Ethernet ports on the sNIC.

We use the following notations. NFs are indexed as $\{1, \cdots, k\}$, VMs as $\{k+1, \cdots, m\}$, and sNIC's Ethernet ports as $\{m+1, \cdots, n\}$. Let $t_{i,j}$ denote the weight of edge (i, j), and E be a set of all edges in the graph. A decision variable $d_{i,j}$ is defined as $d_{i,j} = 1$ if $i \in H$, $j \in N$ and $(i, j) \in E$, 0 otherwise. Let $p_i = 1$ if $i \in H$ and 0 otherwise. Since VMs are always deployed on the hypervisor, and Ethernet ports are on the sNIC, we have $p_i = 1$ if $i \in \{k+1, \cdots, m\}$ and $p_i = 0$ if $i \in \{m+1, \cdots, n\}$. To better leverage the hardware acceleration, $p_i = 1$ if NF i can be accelerated by sNIC. T and D represent the maximum bandwidth of the PCIe bus, and the sNIC's maximum resource capacity, respectively. Each NF i will consume h_i and n_i units of resources when deployed on the host hypervisor and the sNIC, respectively. As an NF's resource requirement depends on the traffic throughput it handles, we have $h_i = r_{host} \sum_{i=1}^{n} t_{i,j}$ and $n_i = r_{nic} \sum_{i=1}^{n} t_{i,j}$, where r_{host} and r_{nic} are NF-specific constants that capture the relationship between the hypervisor's/sNIC's resource requirement and traffic. If an NF instance i can leverage sNIC's hardware acceleration, its n_i will be significantly lower than h_i. While we model the resource requirement as an one-dimensional attribute, the formulation can be generalized for multiple resources (e.g., CPU, memory). Based on these notations, we formulate the NF placement decision problem as an integer linear programming (ILP) shown in Algorithm 1.

Our objective is to minimize the total resource requirements on the hypervisor, under the constraints that the traffic throughput on the PCIe bus should not exceed T, and that the total resource requirements on the sNIC are limited by D. If an edge (i,j) is selected in a cut, the vertices i and j must be in different partitions. While this ILP is an NP-hard problem, for the problem instance sizes that arise for our application, we can efficiently find the optimal solution using off-the-shelf ILP solvers.

Algorithm 1 NF Placement Decision ILP

$$
\begin{array}{lllll}
\text{minimize} & \sum_{i=1}^{k} p_i h_i & & \\
\text{subject to} & \sum_{(i,j)\in E} d_{i,j} t_{i,j} \leq T, & & \\
& \sum_{j=1}^{k} (1-p_i) n_i \leq D, & & \\
& d_{i,j} & \geq p_i - p_j, & (i,j) \in E \\
& p_i & = 1, & i \in \{k+1, \cdots, m\} \\
& p_i & = 1, & i \in \{1, \cdots, k\}, \\
& & & i \text{ is accelerated by sNIC,} \\
& p_i & = 0, & i \in \{m+1, \cdots, n\} \\
& p_i & \in \{0,1\}, & i \in \{1, \cdots, k\} \\
& d_{i,j} & \in \{0,1\}, & (i,j) \in E
\end{array}
$$



(a) NF placement. (b) NF migration.

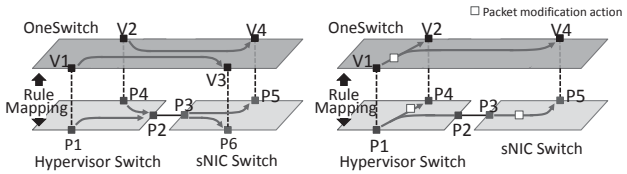
Figure 5: Port/rule/NF mapping in UNO.

If there are k sNICs ($k > 1$) available on the host, the problem becomes a k-way cut problem [15], which can be solved by resource partitioning.

3.2.2 NF placement and migration. The NF agent maintains a topology of NFs and VMs in the host and sNIC, as well as resource requirements (h_i and n_i) of each NF/VM i. When a new NF instance needs to be deployed on the host, NF agent runs the placement algorithm based on the current information. If the remaining resources in the sNIC satisfy the new NF's resource requirement, and the PCIe bandwidth utilization is within T after adding the new NF, we deploy the new NF instance at the sNIC, otherwise at the host hypervisor.

Periodically, NF agent re-runs the algorithm to check the optimality of the placement decision. It initiates NF migration only if the aggregate host resource utilization is far apart from the newly computed solution. The NF state is migrated using the technique presented in [52]. Associated control plane update is described in Section 3.3.2. We are currently investigating an incremental s-t cut problem which can identify the minimal set of migrations needed to meet new inputs/demands. Frequent migration could occur if traffic changes cause oscillations between two configurations. Standard techniques (akin to route dampening) [90] could be applied to prevent frequent migrations; we have not implemented these in our current prototype.

Fig. 5(a) illustrates a case where the NF agent deploys a tenant VM and an IDS-type NF on the hypervisor. After connecting them to the hypervisor switch, the NF agent creates port mappings (V1:P1) and (V3:P5), and notifies the NFV orchestrator that ports V1 and V3 are provisioned for the VM/IDS instances, hiding the



(a) Multiple ingress ports. (b) Multiple egress ports.

Figure 6: Ambiguities in rule translation.

actual ports P1 and P5. Later, when the NF agent decides to migrate an IDS instance to the sNIC due to changing traffic demand, the NF agent triggers port re-mapping, such that the existing port mapping (V3:P5) is updated to (V3:P6) (Fig. 5(b)). The management plane remains unchanged before and after IDS migration as the IDS remains logically connected to V3.

3.3 OneSwitch

OneSwitch hides the hypervisor and sNIC switches and their control interfaces from the data center-wide SDN control plane. It constructs a single virtual data plane using the virtual ports created by the NF agent, and exports this virtual data plane to the controller. When a rule r is pushed to the virtual data plane by the controller, OneSwitch translates r into a set of rules for the underlying physical data planes, such that r's packet processing logic is semantically equivalent to that of the translated rules. We call the rules pushed by the SDN controller *virtual rules*, and the rules installed on the host/sNIC switches after rule translation *physical rules*.

3.3.1 Rule translation algorithm. We leverage the OpenFlow standard [33] for match-action type rule specification. Let's assume that we have a set of k switches $S = \{s_1, s_2, \cdots, s_k\}$ connected to OneSwitch. For example, if there is one sNIC on a host, $k = 2$ (one hypervisor switch and one sNIC switch). Given a virtual rule r, the rule translation algorithm produces and installs a set of N physical rules $R = \{r_j^i \mid i \in S', j = 1, 2, ..., N\}$, where $S' \subseteq S$ and r_j^i is a j-th physical rule installed on a switch i. A correct rule translation implies that for any ingress traffic, the rule r and the rule set R produce exactly the same egress traffic.

Port-map based rule translation: We first describe our basic approach to translate a virtual rule into a set of physical rules by using virtual-to-physical port mappings (`port-map`). We define an *ingress port* as an input port specified in a virtual rule's match condition, and an *egress port* as an output port specified in a virtual rule's forward action. A virtual rule can have zero or one ingress port and zero or more egress ports. We call the switch to which an ingress port is mapped an *ingress switch*, and the switch to which an egress port is mapped an *egress switch*. Since a virtual rule's ingress port and egress port can be mapped to two different physical switches (hypervisor and sNIC), we proceed with rule translation as follows: (1) If a virtual rule does not specify any ingress port or egress port, we install the virtual rule directly into all k switches in S; (2) If a virtual rule only specifies an ingress port, but not any egress port, we install the virtual rule into an ingress switch. (3) If a virtual rule specifies both an ingress port and egress port(s), then for each (ingress port, egress port) pair,

we construct a routing path from an ingress switch to an egress switch. We install a forwarding rule on the ingress switch and each intermediate switch along the path. At the egress switch, we install a forward rule with any other non-forward actions found in the original virtual rule. (4) If a virtual rule only specifies egress port(s), but not any ingress port (i.e., wild card in terms of input port), we first convert the rule into a union of multiple rules with a specific ingress port, and translate each such rule by following step (3).

Pitfalls and solutions: While the above `port-map` based rule translation may seem straightforward, ambiguity can arise when multiple virtual rules co-exist on the virtual data plane. In particular, two possible sets of issues can arise due to "multi-ingress" and "multi-egress" rule translations, which are illustrated respectively in Fig. 6. Fig. 6(a) shows two virtual forwarding rules: (V1→V3) and (V2→V4). The first rule (V1→V3) can be translated to two physical rules (P1→P2) and (P3→P6). However, translation of the second rule (V2→V4) leads to ambiguity on the sNIC switch, as ingress traffic on P3 has two conflicting actions: forward to P5 and forward to P6. A simple ingress port-based match condition cannot disambiguate traffic destined to more than one egress ports. To disambiguate this "multi-ingress" rule translation, we introduce new actions to tag/untag traffic. That is, we apply a *push-flow-id(f)* action at an ingress switch, use the flow-id f as a match condition at an egress switch, and apply a *pop-flow-id* action before any other action. More broadly, the flow-id f encodes the metadata-based flow match conditions (e.g., ingress port, table-id, register value) that cannot be carried across different switches. Tagging traffic with flow-id allows such match conditions to be carried from an ingress switch to an egress switch (if they are different). For simplicity, we generate flow-id f from *hash*(match conditions) at an ingress switch.

Fig. 6(b) shows the situation where traffic tagging/untagging is not sufficient. Here, the virtual rule has match conditions (in-port=V1, ipv4-src=10.0.0.1) and actions (mod-ipv4-src=1.1.1.1, output=V2, output=V4). Since ingress port V1 and egress port V4 are located in two different switches, we need to apply *push-flow-id* action on the ingress switch before traffic exits the switch. However, the problem is that another egress port V2 is mapped to the same switch as ingress port V1. Thus *push-flow-id* action should not be applied when traffic is forwarded to V2, which is mapped to P4. To address this "multi-egress" translation conflict, we apply *push-flow-id* and forward actions *in two stages* through an extra rule table designated X. The rule translation results of these two scenarios are found in Table 1.

To the best of our knowledge, these are the only ambiguities. The final rule translation algorithm can handle the aforementioned ambiguities arising from multi-ingress/egress rules, but is omitted due to the space limitations. The detailed algorithm can be found in [42].

3.3.2 Port remapping and loss free NF migration. In UNO, the NF migration must satisfy two requirements. It needs to be done transparently without involving the SDN controller, and without incurring packet loss during migration. Also, during migration it is important to ensure all in-flight packets are processed, and updates to NFs' internal state due to such packets are correctly reflected at the NF instance's new location [52].

Virtual rule #1 (Fig. 6(a))		
Switch	Match conditions	Actions
OneSwitch	in-port=V1	output=V3
Translated physical rules		
Hypervisor	in-port=P1	push-flow-id=100, output=P2
sNIC	in-port=P3, flow-id=100	pop-flow-id, output=P6

Virtual rule #2 (Fig. 6(b))		
Switch	Match conditions	Actions
OneSwitch	in-port=V1, ipv4-src=10.0.0.1	mod-ipv4-src=1.1.1.1, output=V2, output=V4
Translated physical rules		
Hypervisor	table-id=0, in-port=P1, ipv4-src=10.0.0.1	push-flow-id=200, output=P2, goto-table=X
	table-id=X, in-port=P1, ipv4-src=10.0.0.1	pop-flow-id, mod-ipv4-src=1.1.1.1, output=P4
sNIC	table-id=0, in-port=P3, flow-id=200, ipv4-src=10.0.0.1	goto-table=X
	table-id=X, in-port=P3, flow-id=200, ipv4-src=10.0.0.1	pop-flow-id, mod-ipv4-src=1.1.1.1, output=P5

Table 1: Flow rule translations.

To maintain the transparency, we rely on port remapping. When an NF instance is to be migrated between the host to the sNIC, OneSwitch re-programs the hypervisor/sNIC switches accordingly. Suppose we want to migrate an old NF at port `X` of switch i to a new NF at port `Y` of switch j, when port `X` is mapped to a virtual port `U` at OneSwitch. Let R_U be a set of virtual rules whose match conditions or actions are associated with port `U`. Once NF migration is initiated, the NF agent first provisions port `Y` at switch j, and connects a new NF at port `Y`.

The NF agent then migrates NF state from the old NF to the new NF. The NF agent triggers OneSwitch to re-map the virtual port `U` to port `Y` at switch j, re-translate R_U based on the new port mapping, and install translated rules but with higher priority than the old rules. Finally, OneSwitch removes all old rules translated from R_U, installs the translated rules with the same priority as R_U, and removes the higher priority ones. These steps ensures that no packets will be dropped during the migration.

However, some packets may be in-flight or arrive after state migration starts. In-flight packets are allowed to complete, but newly arrived packets are buffered at the old NF until migration completes, when they are transferred to the new NF.[2] UNO reduces the latency of buffering using techniques from OpenNF [52]. After the NF state is migrated, buffered packets are processed both at the old NF, for low latency, and at the new NF, to ensure its state is correct. After processing at the new NF, though, the packets are dropped so only one copy of the packet is sent to the next service in the chain.

[2]Modern sNICs have sufficient memory to hold the transient packets. For example, the experimental sNIC [28] used in our prototype comes with 8GB, which can buffer up to 8 seconds of packets at line rate.

3.4 Use Cases

Besides offloading NFs, UNO can be leveraged for several other interesting offload scenarios as described below.

Flow rules offload: Flow rule offload is motivated by the increasing number of fine-grained management policies employed in data centers (e.g., for access control, rate limiting, monitoring, etc.) and resulting CPU overhead [74]. One example of offloadable rules is flow-counting monitoring rules because they are *decoupled* from routing/forwarding rules which may be tied to tenant applications running on the hypervisor [96]. With UNO, one can partition monitoring rules into the hypervisor switch and sNIC switch, while keeping a unified northbound control plane that combines flow statistics from the hypervisor and sNIC switches. Furthermore, sNICs like Mellanox TILE-Gx provide unique opportunities to parallelize flow rule processing on multi-cores via fully programmable hardware-based packet classifiers, and maintain flow tables with a large number of rules in memory [32].

Multi-table offload: Modern SDN switches like OVS support pipelined packet processing via multiple flow tables. Multi-table support enables modularized packet processing pipeline, by which each flow table implements a logically separable function (e.g., filtering, tunneling, NAT, routing). This also helps avoid cross-product rule explosion. However, a long packet processing pipeline comes with the cost of increased per-packet table lookup operations. While OVS addresses the issue with intelligent flow caching [81], a long pipeline cannot be avoided with caching if the traffic profile changes frequently. In this environment, some of the tables can be offloaded to the sNIC switch if the inter-switch PCIe communication can carry any metadata exchanged between split flow tables [33]. Table offloading will be particularly beneficial if there are heavy hits by ingress flows on offloaded table(s) (e.g., ACL table). However, it requires consistent flow rule updates across switches (a known problem for SDNs in general [64]), and care that the offloaded flow table fits in the sNIC's memory.

Systematic hardware offload chaining: Data centers often require traffic isolation through encapsulation (e.g., VxLAN, Geneve, GRE) and heavy-duty security or compression operations (e.g., IPsec, de-duplication). These operations may be chained one after another, e.g., VxLAN encapsulation followed by IPsec. While tunneling, crypto and compression operations are well supported in software, they could impose high CPU overhead. Alternatively, one can leverage hardware offloads available in commodity NICs (e.g., large packet aggregation or segmentation (LRO/LSO), tunneling offload) or standalone hardware assist cards (e.g. Intel QAT [10]) which can accelerate crypto and compression operations over PCIe.

However, pipelining these offload operations presents new challenges, not only because simple chaining of hardware offloads leads to multiple PCIe bus crossings/interrupts, but also because different offloads may stand at odds with one another when they reside on separate hardware. For example, a NIC's VxLAN offload cannot be used along with crypto hardware assistance as it does not work in the request/response mode as crypto offload [24]. Also, segmentation on IPsec's ESP packets is often not supported in hardware, necessitating software-based large packet segmentation before crypto hardware assist. All these restrictions lead to under-utilization of individual hardware offload capacities. Many sNICs are equipped with not only general-purpose cores but also integrated hardware circuitry for crypto, compression operations and tunnel processing. This makes them an ideal candidate for a unified, PCIe-efficient hardware and software offload pipeline, fully programmable under the control of UNO.

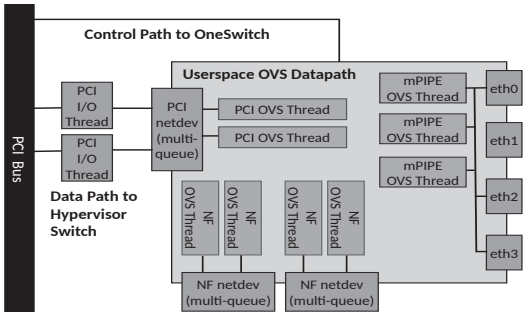


Figure 7: sNIC switch implementation for TILE-Gx.

4 IMPLEMENTATION

We have prototyped the UNO architecture using Mellanox TILE-Gx36 [28] as sNIC, which comes with 36 1.2 GHz CPU cores and four 10GbE interfaces. In this section, we describe key aspects of our implementation.

4.1 NF Agent and OneSwitch

The NF agent exports APIs via which a centralized NFV platform can provision VM/NF instances and their port interfaces on a given host. This northbound interface largely borrows from the OpenStack Compute APIs [20]. Internally, the NF agent uses the CPLEX Python solver [27] to compute optimal NF placements (Algorithm 1) from the current NF traffic workload (NF-level traffic matrix). The current workload is estimated by querying hypervisor/sNIC switches for port/flow statistics. When NF migration is needed, NF agent triggers port remapping in OneSwitch via RESTful APIs and migrates NF state as described in Section 3.2.2.

OneSwitch implementation is based on OpenVirteX (OVX) network virtualization software [34], which can perform basic control plane translation for network slicing. The original OVX implementation is unable to handle rule translations that involve multi-ingress/egress rules illustrated earlier, and does not support dynamic port/rule remapping for NF migration. We extend OVX to incorporate the more general rule translation algorithm described in Section 3.3.1, and dynamic migration support as described in Section 3.2.2.

4.2 Hypervisor/sNIC Switches

In UNO architecture, hypervisor and sNIC switches are regular SDN switches controlled by OneSwitch, and thus we base their implementation on OVS. While the control plane interface of OVS is sufficient for UNO, the unique deployment environment for hypervisor/sNIC switches brings up the following challenges in their data plane implementation: **(C1)** They should support efficient data path spanning across PCIe bus and multiple process boundaries between the switches and NFs; **(C2)** sNIC typically has less per-core compute capacity than x86 host, and in order to support NF

migration between two platforms, the per-port TX/RX processing capacity of NF ports need to be reasonably matched between two switches; **(C3)** sNIC switch should be able to leverage any hardware acceleration available in sNIC.

To address **(C1)**, we leverage kernel-bypass networking, i.e, poll-mode, userspace OVS datapath for both switches, which can eliminate interrupt overheads associated with PCIe bus crossings and avoid memory copies while forwarding to userspace NFs. Currently we dedicate cores to polling, but a future implementation could reduce load by automatically switching to interrupts or coalescing multiple ports onto a single core, similar to how the Linux NAPI framework switches between polling and interrupts. On the x86 host side, we re-use the DPDK OVS datapath, but extend it by adding a PCIe-type `netdev` port and its polling thread. On TILE-Gx side, we implement a custom DPIF provider [6] plugged into userspace OVS, and dedicated PCIe-type and NF-type `netdev` ports. The custom DPIF implementation exploits TILE-Gx mPIPE's hardware-based packet classification and `flow_hash` computation to accelerate data plane processing **(C3)**. To transfer directly between TILE-Gx userspace OVS and x86 host user space OVS via the PCIe-type port pair, we leverage `mmap` on x86 host side, which maps the PCIe DMA buffer allocated by the host PCIe driver into the host userspace, and use zero copy APIs on TILE-Gx side for packet transfer between TILE-Gx memory and the PCIe link. The port pair of two OVS instances is interconnected over PCIe bus via four parallel PCIe packet queues. The resulting data plane design allows line rate traffic to be forwarded from TILE-Gx's Ethernet ports all the way to x86 host userspace.

For scalable TX/RX rates for NF ports **(C2)**, we support a configurable number of TX/RX queues for each NF-type port, which can be determined during port provisioning. Each TX/RX queue is lock-free multi-producer, multi-consumer FIFO queue implementation, and carries packets stored in memory shared between the userspace datapath and NF instances. The TILE-Gx userspace switch implementation is shown in Fig. 7.

To support rules generated by the UNO's rule translation algorithm, both switches need to handle per-packet *flow-id* metadata, and perform *flow-id* based flow matching and push/pop-flow-id actions. We re-purpose VLAN id to store *flow-id* metadata, and leverage corresponding OpenFlow support (i.e., OXM_OF_VLAN_VID match field and push/pop-VLAN actions). Note that the VLAN tagging occurs transparently between the physical OVS and OneSwitch, and is not visible outside the host.

4.3 Network Functions

The aforementioned challenges **(C1)**, **(C2)** and **(C3)** are not unique to switch data plane design, but also relevant to NF implementation. For **(C1)**, an NF leverages userspace poll-mode, shared memory based port interface to exchange traffic with OVS. For multi-core scalability **(C2)**, multiple NF instances can run (one per core), each with a dedicated TX/RX queue for the port. NF implementation also incorporates NF-specific acceleration (e.g., for crypto and compression) using TILE-Gx's built-in accelerator **(C3)**.

For both x86 and TILE-Gx platforms, we implement two custom NFs and modify one existing NF. For custom NFs, we implement layer-7 firewall (L7FW) and IPsec security gateway (SECGW). L7FW detects layer-7 application protocols (e.g., FaceBook, Skype) and selectively blocks them using nDPI [16]. SECGW performs encryption and authentication on clear-text traffic in IPsec ESP tunnel mode [65]. On TILE-Gx side, SECGW leverages MiCA acceleration for IPsec processing. We also modify PRADS [22], a DPI-based asset monitor, so that it can import/export its state for dynamic migration.

The sNIC/hypervisor switches and NFs are developed and extended with C in 22K and 5K SLOC, respectively.

5 EVALUATION

We evaluate the UNO prototype on a server with 24 Intel Xeon 2.7GHz CPU cores and 128GB memory running Ubuntu 13.10 with Linux kernel version 3.11, and use Mellanox TILE-Gx36 specified in Section 4 as sNIC.

5.1 Benefit of Offloading

We evaluate the benefit of sNIC offload to show that using sNICs can help improve CPU utilization, system energy, and I/O bus utilization.

Packet switching offload: In this experiment, we demonstrate the benefit of packet switching offload in terms of host CPU usage. We set up a UNO server with Mellanox TILE-Gx as the sNIC, and two interconnected OVS datapaths deployed on x86 hypervisor and sNIC. We inject traffic at the TILE-Gx's Ethernet port from another server. We consider three scenarios using 64K filtering rules: (1) all 64K rules installed in the hypervisor OVS ("HOST"), (2) half of the rules offloaded to TILE-Gx OVS ("HALF"), (3) all the rules offloaded to TILE-Gx OVS ("sNIC"). In Fig. 8, we report the CPU usage of hypervisor OVS per minute. We can see that the x86 CPU usage is the highest when all the rules are installed in the x86 host, halved by offloading half of the flows to TILE-Gx, and lowest when all the rules are offloaded to TILE-Gx. The number of TILE-Gx CPU cores allocated for sNIC OVS is 12, meaning that there are 24 cores available for other processing. This experiment shows that offloading packet switching to sNIC can effectively release host CPU resources.

Network function offload: In the next set of experiments, we demonstrate the benefit of sNIC for offloading NFs. For this we use two custom NFs we develop: (i) layer-7 firewall (L7FW) (ii) IPsec security gateway (SECGW). In sNIC-side implementation, L7FW runs on sNIC's general-purpose cores, while SECGW is further accelerated with sNIC's built-in crypto engine (TILE-Gx MiCA [32]). In x86 host-side implementation, L7FW is purely software implementation, while SECGW exploits either x86's extended instruction set (Intel AES-NI [7]) or a standalone crypto acceleration card (Intel QAT [10]). Depending on where it is deployed, an NF is connected to either the hypervisor OVS or sNIC OVS.

Energy efficiency: Figs. 11(a) and (b) show the power consumption of L2FW and SECGW, respectively, when they are deployed on either x86 host or sNIC. Traffic is injected from another server, forwarded via sNIC to x86 host, and consumed within the x86 host. The reported power consumption on y-axis is the *increase* in the overall server's power usage (measured with a wattmeter [31]) when the injected traffic increases from zero to the amount shown on x-axis. Due to the difference in single-core processing capability

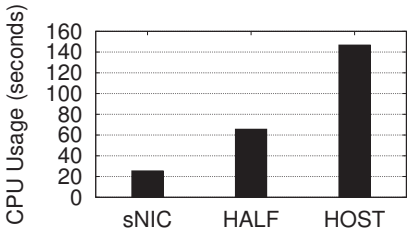


Figure 8: Effect of switching offload on CPU usage.

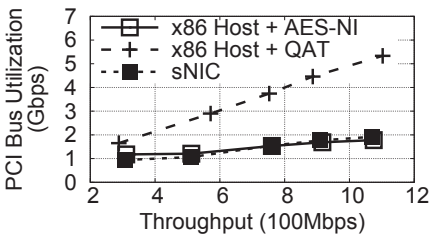


Figure 9: PCIe bandwidth utilization.

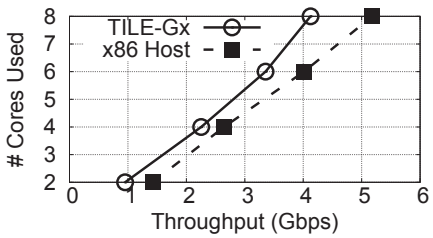
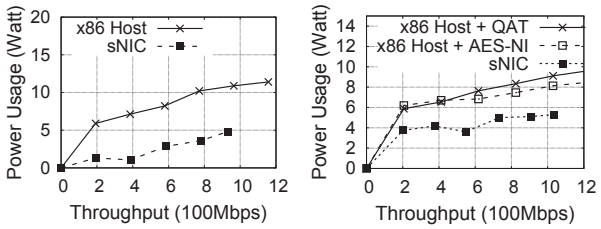


Figure 10: SD-WAN CPU usage.



(a) Layer-7 firewall. (b) IPsec security gateway.

Figure 11: Energy efficiency of sNIC.

between x86 and sNIC, we allocate one core for a host-side NF, and four cores for an sNIC-side NF. The main observations from Fig. 11 are as follows. sNIC's general purpose cores significantly outperform the x86 host's in terms of energy efficiency (e.g., by a factor of 2–3 for L7FW). Even when dedicated hardware acceleration is available for an NF for either platform (AES-NI/QAT on x86 host or MiCA on sNIC), sNIC's network function processing still consumes less energy than that of the x86 host (e.g., by a factor of 1.5–2 for SECGW).

PCIe bus utilization: In Fig. 9, we compare the server's PCIe bus utilization (measured with Processor Counter Monitor [23]) in several SECGW deployment scenarios: (i) sNIC, (ii) x86 host with AES-NI, (iii) x86 host with QAT acceleration. Clear-text UDP packets of 1280 byte size are generated within the x86 host, processed by SECGW, and sent out to the wire as IPsec packets. Compared to sNIC and x86/AES-NI deployment, the PCIe bandwidth usage with x86/QAT deployment is more than doubled. That is because each packet incurs an additional request/response transaction with QAT across PCIe [24]. sNIC deployment is more PCIe bandwidth-efficient than x86/AES-NI deployment because the latter case needs to sustain additional PCIe bandwidth overhead of egress IPsec packets (with IP tunnel header, ESP, padding, etc.). We expect that this benefit of sNIC deployment will become more pronounced with smaller packets.

Host CPU savings: To demonstrate the host CPU saving in realistic scenarios, we deploy a software-defined WAN (SD-WAN) use case [26] on our testbed server, where encrypted packets are injected to SECGW, and SECGW decrypts and forwards packets to L7FW NFs. In one case, we realize a vanilla implementation (userspace, poll-mode) of SD-WAN on the x86 host without sNIC, while in the other case, we deploy the SD-WAN in a UNO setup with sNIC. In both cases, the deployed SECGW and L7FW NFs are scaled up with traffic by adding more x86 CPU cores or TILE-Gx CPU cores. Fig. 10 plots the number of x86 CPU cores or TILE-Gx CPU cores required to support 10K UDP flows of a particular throughput. For example, to support 4Gbps throughput, we need to allocate 8 TILE-Gx CPU cores in UNO deployment, or 6 x86 CPU cores in x86 host deployment. If UNO is adopted for the SD-WAN application, the number of x86 CPU cores are saved and can be used for tenant applications by offloading to TILE-Gx CPUs. Note also that UNO enables SECGW to be offloaded from TILE-Gx CPUs to the built-in crypto engine. This allows SD-WAN offload to achieve significant host CPU reduction even with a small number of TILE-Gx CPU cores with limited compute capacity. We are currently investigating why bandwidth tops out at 4Gbps for the sNIC and believe it is related to the Tilera OVS implementation.

5.2 Cost of NF Migration

If traffic demand is higher than 4Gbps, an NF migration is required, i.e., either SECGW or L7FW will need to be migrated to the x86 host. In this section, we evaluate the cost of such NF migration. For this we run an experiment with PRADS [52], where the NF agent performs loss-free migration of PRADS from the x86 host to the sNIC while the NF is processing traffic. During migration, the old NF on x86 host serializes its per-flow states and transfers them to the NF agent, which then transfers the states to the new NF on sNIC, where the states are deserialized.

Fig. 13 shows how the migration time scales with the number of flows affected. Serialization and deserialization occur concurrently; migration refers to the remaining time overhead. As expected, the NF migration overhead increases with the number of flows because the state size increases. Note that the flow state size does not linearly increase with the number of flows. In this experiment, the flow state size is 1.4MB for 750 flows, and 1.6MB for 1000 flows. Deserialization on sNIC takes longer than serialization on x86 host due to the lower single-core performance of sNIC. The trend becomes the opposite when migrating from sNIC to x86 host. We also measure per-packet latency during the NF migration period when ingress packets are temporarily buffered and forwarded by the NF agent. We find that average per-packet latency increases by 40–50ms during migration, compared to migration-free condition. The current NF migration scheme can be improved with possible alternatives. For example, we can leverage serialization-free, memory-mapped state transfer

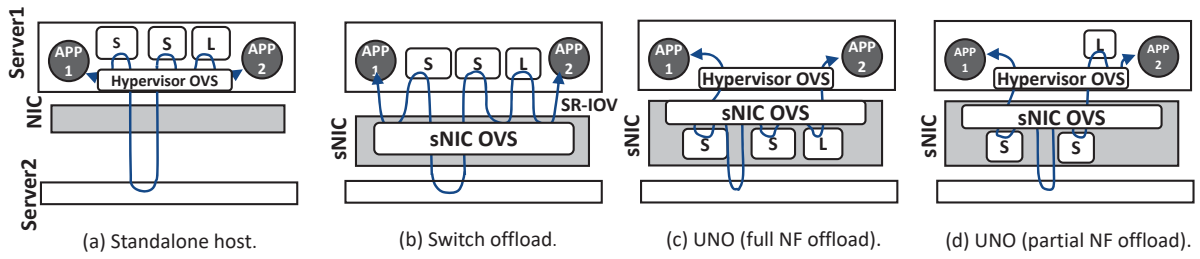


Figure 12: Latency experiment setup. 'S' indicates SECGW, and 'L' indicates L7FW.

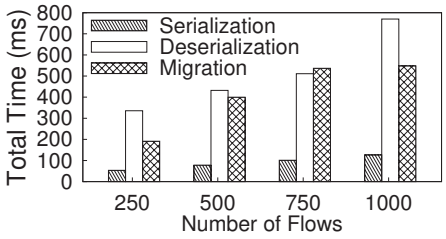


Figure 13: NF migration overhead.

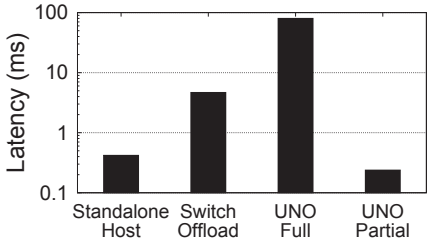


Figure 14: Latency.

between old and new NFs over PCIe [66], where NF agent simply signals the state transfer. Another possibility is to decouple NF states from NFs [61], and move them to a separate in-memory storage shared between the x86 host and the sNIC, which also obviates expensive serialization. We leave further improvement on NF migration for future work.

5.3 Effect of NF Placement

The previous results showed that while offloading functions can save energy and reduce x86 host utilization, some traffic patterns can actually worsen throughput and latency. This motivates the need for good decisions on where to place network functions. In the next experiment, we evaluate UNO's ability to make good NF placement decisions. The algorithm runs as part of the NF Agent, and executes efficiently on an commodity x86-based servers (e.g., 0.06 second with 100 nodes and 1 second with 500 nodes). First, we measure the capability of sNIC's CPU cores vs. x86 CPU cores using the ratio n_i / h_i on a set of NFs, i.e., DPI, L7FW and SECGW; this ration is provided as input to the ILP framework to make the NF decisions. We find that the ratios for DPI, L7FW and SECGW are 2, 4, and 0.75, respectively.

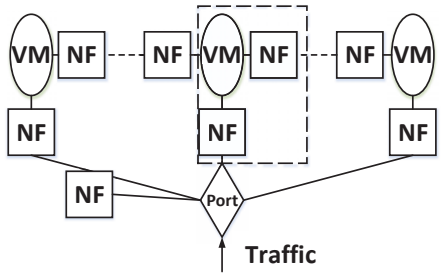
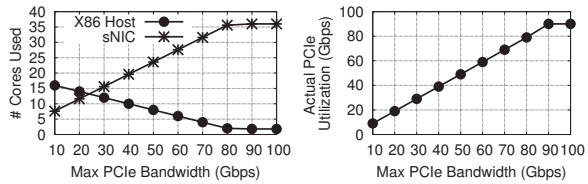


Figure 15: Placement graph as defined in Section 3.2.1: The dotted line rectangle part is repeated 7 times along the dotted line.

Placement decision: In this simulation, we invoke UNO's placement algorithm directly to evaluate its decisions. We use one NF type—DPI for easy demonstration—and the placement graph of NF/VM instances as shown in Fig. 15, and set the number of VM and NF instances in the graph to 9 and 18, respectively, following typical VM workload-to-server ratios in public cloud data centers [56]. We set the total sNIC resource capacity D to 36 to reflect the sNIC's total core count, and vary the maximum PCIe throughput T. We set the resource requirements h_i and n_i of NF instances based on the coarse-grained resource profiling results earlier which indicated that a DPI function requires 5 cores on the x86 host and 10 cores on sNIC to process 9Gbps of incoming traffic. Figs. 16(a) and (b) shows how UNO trades off between x86 host core usage, sNIC core usage, and PCIe bandwidth usage. We can see that when the maximum PCIe bandwidth is small (10Gbps), it is the bottleneck and UNO puts the NFs to the x86 host using more CPU cores. When the maximum PCIe bandwidth increases, UNO offloads more NFs to the sNIC, thus decreasing host CPU utilization while increasing sNIC CPU utilization. This trend holds on until sNIC's full capacity is reached, when the maximum PCIe bandwidth exceeds 80Gbps. This experiment demonstrates that the UNO's placement algorithm effectively leverages available PCIe bandwidth and sNIC's compute capacity to offload as much NF workload as possible.

Network latency: UNO seeks to minimize host CPU utilization, but the optimal placement can negatively affect latency. To illustrate this, we set up a NF chain of two SECGWs and one L7FW, as shown in Fig. 12 and measure the round trip latency between APP1 and APP2 as shown in Fig. 14. The two SECGWs act as IPsec tunnel



(a) x86 CPU utilization. (b) PCIe utilization.

Figure 16: Simulation-based DPI NF placements.

endpoints between APP1 and APP2. Fig. 12(a) is the deployment of the chain on a standalone x86 host with a regular NIC. In Fig. 12(b), only the packet switching is offloaded to sNIC, with all NFs running on the x86 host. For traffic demand of 1Gbps, UNO will offload all network functions (UNO Full) as shown in Fig 12(c). This results in zero host CPU usage, but high latency, i.e.,79ms, compared to either standalone host, or switch offload, because the weaker sNIC CPU runs much slower than the x86 CPU for the L7FW function. When the traffic demand is 4Gbps or higher, UNO uses the configuration (UNO-Partial) as shown in Fig 12(d), which uses more host CPUs to handle the increased workload. This has the beneficial side effect of reducing the latency to 0.2ms. This experiment reveals that the UNO's current placement algorithm does not always improve the latency because its objective is to minimize x86 host resource usage. We plan to consider jointly optimizing secondary objectives (latency and throughput) along with CPU resources in future work.

5.4 Flow Rule Translation

UNO's local control plane translation can provide the potential scalability benefit for the external SDN controller, as UNO can hide local sNIC(s) from the SDN controller, relieving the controller of its responsibility for managing sNIC switches. To demonstrate this benefit, we set up a UNO testbed consisting of a Floodlight controller [5], OneSwitch, one hypervisor switch and two sNIC switches. Then we synthetically generate 3,024 virtual rules by randomly choosing one or more match conditions and actions from OpenFlow 1.3, and count the total number of physical rules translated from them. We randomly sample 200 virtual rules, and manually verify the correctness of their translation. In total, 6,810 physical rules are installed on three physical switches. Were these three switches fully exposed to Floodlight without UNO, a similar number of such rules would need to be processed directly by Floodlight, which is a factor of 2.3 increase compared to virtual rules. Conversely, it means that UNO can reduce the controller overhead by that much by distributing sNIC control on to individual end hosts.

6 RELATED WORK

To improve end host networking performance, several different approaches are proposed; purely software solutions [57, 85], SR-IOV based switch offload [37, 79], and hybrid solutions which combine software functions and hardware NICs for performance [86] or flexibility [55]. However, none of these considers full hardware programmability for NF offload. FPGA-based NF acceleration approaches [46, 50] propose a flexible and high-performance hardware-accelerated data plane, but the flexibility comes in the form of FPGA's configurability, not flow-level programmability like UNO. HyperFlow [95] constructs a single logical controller from multiple controllers, which is similar to UNO, but its goal is to provide a scalable control plane.

UNO's multi-switch model requires flow rule translation. There have been several research efforts on SDN flow rule construction, transformation and distribution for realizing higher level network policies. Due to the rule space capacity limits on switches, [73, 74, 99] distribute flow rule tables across the network to enforce endpoint policies such as access control or load balancing, but may change traffic routing paths. [62, 63] strive to optimize rule space utilization by mapping flow rules to dispersed switches while maintaining both endpoint and routing polices. OVX [34] uses flow rule translation to implement network virtualization. While UNO's flow rule translation shares similarity with some of them on maintaining a per node single virtual switch abstraction and thus hiding complexities from SDN controllers, its goal is to optimize the local resource use by leveraging NF placement/migration and sNIC offload capacities.

There is a large body of work that addresses NF placement and migration in a data center environment [48, 51, 52, 76, 82, 89, 91]. The common goal of them is to place-and-chain new NF instances and relocate-and-rechain existing NF instances across multiple hosts. UNO is concerned with NF placement and relocation within a single host (augmented with sNIC). While [60, 72] also empower a host resident OVS to handle NFs within a host, they focus on extending the reach of Openflow rather than intelligent offload as performed by UNO. In principle our approach is similar to prior work on offloading computational tasks from a mobile phone to the infrastructure in a mobile environment [44]. We differ in our objectives (minimizing host CPU resources), the offloading target, and the workloads offloaded.

7 CONCLUSION

In this paper, we presented the design, implementation and evaluation of an SDN-controlled NF offload architecture called UNO. UNO can transparently leverage the smart NIC's programmable compute capabilities to accelerate the NF data plane, and without introducing additional complexity in the data center's centralized management and control planes. UNO's transparent offload is achieved by two per-host components: the NF agent which intelligently chooses a subset of NFs to offload to the sNIC, and OneSwitch which abstracts out the offloaded NF data planes from the data center's control plane. Together, these two components hide the complexity prevalent in local traffic pattern-based dynamic NF offload decisions and the intricacies of NF migration (e.g., internal state management) from the data center controller. The evaluation results demonstrate the feasibility and the substantial benefits of UNO.

Acknowledgments

We would like to thank the reviewers and our shepherd, Noa Zilberman, for their thoughtful feedback. Yanfang Le, Aditya Akella and Michael Swift are supported in part by NSF grants CNS-1636563, CNS-1629198, CNS-1551745, and CNS-1330308, and via gifts from Huawei and Facebook.

REFERENCES

[1] Accolade ANIC. https://accoladetechnology.com/whitepapers/ANIC-Features-Overview.pdf.
[2] Cavium LiquidIO. http://www.cavium.com/pdfFiles/LiquidIO_Server_Adapters_PB_Rev1.0.pdf.
[3] Data Center Market Trends. http://www.te.com/content/dam/te-com/documents/broadband-network-solutions/global/data-center/brochures/presentation-data-center-market-trends.pdf.
[4] Emerging Smart NIC Technology. http://www.csit.qub.ac.uk/News/Events/Belfast-2016-6th-Cyber-Security-Summit/PDFs/Filetoupload,631658,en.pdf.
[5] Floodlight. http://www.projectfloodlight.org/floodlight/.
[6] How to Port Open vSwitch to New Software or Hardware. http://openvswitch.org/support/dist-docs-2.5/PORTING.md.html.
[7] Intel Advanced Encryption Standard (Intel AES) Instructions Set - Rev 3.01. https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set.
[8] Intel Clear Containers: A Breakthrough Combination of Speed and Workload Isolation. https://clearlinux.org/sites/default/files/vmscontainers_wp_v5.pdf.
[9] Intel Gigabit Server Adapters. http://ark.intel.com/products/family/46829.
[10] Intel QuickAssist Adapter Family for Servers. http://www.intel.com/content/www/us/en/ethernet-products/gigabit-server-adapters/quickassist-adapter-for-servers.html.
[11] Max-flow min-cut theorem. https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem.
[12] Mellanox BlueField. http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf.
[13] Mellanox ConnectX-4. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf.
[14] Mellanox ConnectX-5. http://www.mellanox.com/related-docs/user_manuals/ConnectX-5_VPI_Card.pdf.
[15] Minimum k-cut. https://en.wikipedia.org/wiki/Minimum_k-cut.
[16] nDPI. http://www.ntop.org/products/deep-packet-inspection/ndpi/.
[17] Netronome Agilio vRouter. https://netronome.com/media/redactor_files/SB_Netronome_Juniper_vRouter.pdf.
[18] OpenDaylight. https://www.opendaylight.org.
[19] OpenStack. https://www.openstack.org.
[20] OpenStack Compute API. https://developer.openstack.org/api-ref/compute/.
[21] OPNFV. https://www.opnfv.org.
[22] PRADS – Passive Real-time Asset Detection System. https://gamelinux.github.io/prads/.
[23] Processor Counter Monitor. https://github.com/opcm/pcm.
[24] Programming Intel QuickAssist Technology Hardware Accelerators for Optimal Performance. https://01.org/sites/default/files/page/332125_002_0.pdf.
[25] Putting Smart NICs in White Boxes. https://www.sdxcentral.com/articles/analysis/nics-white-boxes/2016/11/.
[26] SD-WAN. https://en.wikipedia.org/wiki/SD-WAN.
[27] Setting up the Python API of CPLEX. http://www.ibm.com/support/knowledgecenter/SSSA5P_12.5.1/ilog.odms.cplex.help/CPLEX/GettingStarted/topics/set_up/Python_setup.html.
[28] TILEncore-Gx36. http://www.mellanox.com/related-docs/prod_multi_core/PB_TILEncore-Gx36.pdf.
[29] Tilera Rescues CPU Cycles with Network Coprocessors. https://www.enterprisetech.com/2013/10/16/tilera-free-expensive-cpu-cycles-network-coprocessors/.
[30] VMware. Data Center Micro-Segmentation. http://blogs.vmware.com/networkvirtualization/files/2014/06/VMware-SDDC-Micro-Segmentation-White-Paper.pdf.
[31] Watts Up Meter. https://www.wattsupmeters.com.
[32] TILE Processor Architecture Overview for the TILE-Gx Series. Technical report, Mellanox, 2012. Doc. No. UG130.
[33] OpenFlow Switch Specification 1.5.0. Open Network Foundation, 2014.
[34] A. Al-Shabibi et al. OpenVirteX: Make Your Virtual SDNs Programmable. In *Proc. ACM HotSDN*, 2014.
[35] S. P. Antoine Kaufmann and N. K. Sharma. High Performance Packet Processing with FlexNIC. In *Proc. ASPLOS*, 2016.
[36] H. Ballani et al. Enabling End-host Network Functions. In *Proc. ACM SIGCOMM*, 2015.
[37] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. USENIX OSDI*, 2014.
[38] M. Blott and K. Vissers. Dataflow Architectures for 10Gbps Line-rate Key-value-Stores. In *Proc. IEEE Hot Chips 25 Symposium*, 2013.
[39] P. Bosshart et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3), 2014.
[40] Z. Bozakov and P. Papadimitriou. AutoSlice: Automated and Scalable Slicing for Software-Defined Networks. In *Proc. ACM CoNEXT*, 2012.
[41] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Proc. ACM HotSDN*, 2012.
[42] H. Chang, S. Mukherjee, L. Wang, T. Lakshman, Y. Le, A. Akella, and M. Swift. UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing. Technical Report ITD-16-56788B, Nokia, 2016.
[43] Cisco. Data Center Microsegmentation: Enhance Security for Data Center Traffic. http://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-732943.html.
[44] E. Cuervo et al. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. ACM MobiSys*, 2010.
[45] H. T. Dang et al. Network Hardware-Accelerated Consensus. In *USI Technical Report Series in Informatics*, 2016.
[46] R. R. David F. Bacon and S. Shukla. FPGA Programming for the Masses. *ACM QUEUE*, 11(2), 2013.
[47] W. Dietz, J. Cranmer, N. Dautenhahn, and V. Adve. Slipstream: Automatic Interprocess Communication Optimization. In *Proc. USENIX ATC*, 2015.
[48] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Proc. USENIX NSDI*, 2014.
[49] D. Firestone. SmartNIC: Accelerating Azure's Network with FPGAs on OCS Servers. Open Compute Project, 2016.
[50] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu. OpenANFV: Accelerating Network Function Virtualization with a Consolidated Framework in OpenStack. In *Proc. ACM SIGCOMM*, 2014.
[51] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward Software-defined Middlebox Networking. In *Proc. ACM HotNets-XI*, 2012.
[52] A. Gember-Jacobson et al. OpenNF: Enabling Innovation in Network Function Control. *ACM SIGCOMM Computer Communication Review*, 44(4), 2015.
[53] B. Grot et al. Optimizing Data-Center TCO with Scale-Out Processors. *IEEE Micro*, 32(5), 2012.
[54] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network Functions Virtualization: Challenges and Opportunities for Innovations. *IEEE Communication Magazine*, 53(2), 2015.
[55] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, University of California, Berkeley, 2015.
[56] A. Holt et al. Cloud Computing Takes Off. https://www.morganstanley.com/views/perspectives/cloud_computing.pdf. Morgan Stanley.
[57] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. mSwitch: A Highly-Scalable, Modular Software Switch. In *Proc. ACM SOSR*, 2015.
[58] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms. In *Proc. USENIX NSDI*, 2014.
[59] Z. Istvan, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proc. USENIX NSDI*, 2016.
[60] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker. SoftFlow: A Middlebox Architecture for Open vSwitch. In *Proc. USENIX ATC*, 2016.
[61] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *Proc. USENIX NSDI*, 2017.
[62] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the i£¡One Big Switchi£¡ Abstraction in Software-Defined Networks. In *Proc. ACM CoNEXT*, 2013.
[63] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing Tables in Software-Defined Networks. In *Proc. ACM CoNEXT*, 2013.
[64] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.
[65] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303, 2005.
[66] A. Khrabrov and E. de Lara. Accelerating Complex Data Transfer for Cluster Computing. In *Proc. USENIX HotCloud*, 2016.
[67] Kindervag, J. Build Security Into Your Network's DNA: The Zero Trust Network Architecture.
[68] S. Larsen and B. Lee. Platform IO DMA Transaction Acceleration. In *Proc. ACM Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, 2011.
[69] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proc. USENIX OSDI*, 2016.
[70] K. Lim et al. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proc. ISCA*, 2013.
[71] Y. Luo, E. Murray, and T. L. Ficarra. Accelerated Virtual Switching with Programmable NICs for Scalable Data Center Networking. In *Proc. ACM VISA*, 2010.
[72] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman. Application-aware Data Plane Processing in SDN. In *Proc. ACM HotSDN*, 2014.
[73] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vCRIB: Virtualized Rule Management in the Cloud. In *Proc. USENIX HotCloud*, 2012.
[74] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable Rule Management for Data Centers. In *Proc. USENIX NSDI*, 2013.

[75] J. Nam, M. Jamshed, B. Choi, D. Han, and K. Park. Scaling the Performance of Network Intrusion Detection with Many-core Processors. In *Proc. ACM/IEEE ANCS*, 2015.
[76] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proc. ACM SOSP*, 2015.
[77] Palo Alto Networks. Getting Started With a Zero Trust Approach to Network Security. https://www.paloaltonetworks.com/resources/whitepapers/zero-trust-network-security.html.
[78] T. Park, Y. Kim, and S. Shin. UNISAFE: A Union of Security Actions for Software Switches. In *Proc. SDN-NFV Security*, 2016.
[79] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proc. USENIX OSDI*, 2014.
[80] J. Pettit. Open vSwitch and the Intelligent Edge. In *Proc. OpenStack Summit Atlanta*, 2014.
[81] B. Pfaff et al. The Design and Implementation of Open vSwitch. In *Proc. USENIX NSDI*, 2015.
[82] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. ACM SIGCOMM*, 2013.
[83] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *Proc. USENIX NSDI*, 2014.
[84] B. Raghavan et al. Software-Defined Internet Architecture: Decoupling Architecture from Infrastructure. In *Proc. ACM HotNets-XI*, 2012.
[85] K. K. Ram, A. L. Cox, M. Chadha, and S. Rixner. Hyper-switch: A scalable software virtual switching architecture. In *Proc. USENIX ATC*, 2013.
[86] K. K. Ram et al. sNICh: Efficient Last Hop Networking in the Data Center. In *Proc. ACM/IEEE ANCS*, 2010.
[87] L. Rizzo, P. Valente, G. Lettieri, and V. Maffione. PSPAT: software packet scheduling at hardware speed. Preprint, 2016.
[88] G. Sabin and M. Rashti. Security Offload Using the SmartNIC, A Programmable 10 Gbps Ethernet NIC. In *Proc. Aerospace and Electronics Conference*, 2015.
[89] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. USENIX NSDI*, 2012.
[90] A. Shaikh, J. Rexford, and K. G. Shin. Load-Sensitive Routing of Long-Lived IP Flows. In *Proc. ACM SIGCOMM*, 1999.
[91] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proc. ACM SIGCOMM*, 2012.
[92] R. Sherwood et al. FlowVisor: A Network Virtualization Layer. In *OpenFlow Switch Consortium*, 2009.
[93] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle. We need to talk about NICs. In *Proc. USENIX HotOS*, 2013.
[94] D. Sturgeon. HW Acceleration of Memcached. In *Proc. Flash Memory Summit*, 2014.
[95] A. Tootoonchian and Y. Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proc. Internet Network Management Conference on Research on Enterprise Networking*, 2010.
[96] A. Wang, Y. Guo, F. Hao, T. V. Lakshman, and S. Chen. UMON: Flexible and Fine Grained Traffic Monitoring in Open vSwitch. In *Proc. ACM CoNEXT*, 2015.
[97] Z. Wang, K. Liu, Y. Shen, J. Y. B. Lee, M. Chen, and L. Zhang. Intra-host Rate Control with Centralized Approach . In *Proc. IEEE International Conference on Cluster Computing*, 2016.
[98] Y. Weinsberg, D. Dolev, P. Wyckoff, and T. Anker. Accelerating Distributed Computing Applications Using a Network Offloading Framework. In *Proc. IEEE Parallel and Distributed Processing Symposium*, 2007.
[99] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In *Proc. ACM SIGCOMM*, 2010.