Revisiting the Open vSwitch Dataplane Ten Years Later

William Tu
VMware
United States
tuc@vmware.com

Yi-Hung Wei
VMware
United States
yihungw@vmware.com

Gianni Antichi
Queen Mary University of London
United Kingdom
g.antichi@qmul.ac.uk

Ben Pfaff
VMware Research
United States
bpfaff@vmware.com

ABSTRACT

This paper shares our experience in supporting and running the Open vSwitch (OVS) software switch, as part of the NSX product for enterprise data center virtualization used by thousands of VMware customers. From its origin in 2009, OVS split its code between tightly coupled kernel and userspace components. This split was necessary at the time for performance, but it caused maintainability problems that persist today. In addition, in-kernel packet processing is now much slower than newer options (e.g. DPDK).

To solve the problems caused by the user/kernel split, OVS must adopt a new architecture. We describe two possibilities that we explored, but did not adopt: one because it gives up compatibility with drivers and tools that are important to virtual data center operators, the other because it performs poorly. Instead, we endorse a third approach, based on a new Linux socket type called AF_XDP, which solves the maintainability problem in a compatible, performant way. The new code is already merged into the mainstream OVS repository. We include a thorough performance evaluation and a collection of lessons learned.

CCS CONCEPTS

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

KEYWORDS

virtual switch, eBPF, XDP, OVS

ACM Reference Format:
William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the Open vSwitch Dataplane Ten Years Later. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3452296.3472914

SIGCOMM '21, August 23–27, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472914

1 INTRODUCTION

Compute virtualization is commonplace in data center networks [6]. Communication endpoints are no longer physical servers, but rather virtual machines (VMs) or containers, whose networking is typically managed through a network virtualization system such as VMware NSX [77] or Microsoft Hyper-V [44]. These systems apply high-level policies to incoming packets, forwarding them to physical interfaces, containers, or VMs using a software switch, e.g., VFP [24], Contrail vRouter [35], or Open vSwitch [56].

In this paper we share our experience in supporting Open vSwitch (OVS), which is adopted in many data center networks [1, 21, 50] and which is now a component of VMware's NSX product, used by thousands of customers. When the development of OVS started, one of the design goals was to achieve compatibility with existing software. For instance, OVS was at the time an upstart competitor to the "Linux bridge" [36], which dominated the category on Linux, and thus OVS provided bridge compatibility utilities, daemon, and even an additional kernel module to make migration easy. OVS also maintained compatibility with kernel network device drivers and tools, such as `ip`, `ping`, and `tcpdump`. This kind of compatibility was needed so obviously that it was not even discussed; any other option would have severely curtailed OVS adoption. Bridge compatibility has gone by the wayside, but driver and tool compatibility remains important. For VMware, this is because OVS is the dataplane of NSX, which supports deployment scenarios ranging across many Linux distributions, kernel versions and variants, and physical and virtual hardware.

Another important design goal that we considered was performance. When OVS was originally designed, only the kernel could process packets at high speed, but OVS was too complex to implement entirely inside the kernel. Thus, the original OVS architecture split its code between a userspace process, which set policy, and a kernel module, which provided the least mechanism needed to implement the policy with high performance [56]. The kernel module was tightly coupled both to Linux kernel internal interfaces and to the userspace process. This was the only design that met the performance and feature requirements, a design accepted by both OVS and kernel developers, but the tight coupling led to the following drawbacks, which have caused substantial problems for OVS developers and users:

- ***Maintainability***: Linux and Open vSwitch are separate projects with largely different developers. The tight coupling between the kernel module and userspace means these two groups must come

to one consensus on design and development. Features in OVS are limited by what Linux developers will accept in principle and then in implementation. Section 2.1 will detail these issues.
- ***Operability***: OVS upgrades or bug fixes that affect the kernel module can require updating the kernel and rebooting production systems, disrupting any workloads running there or requiring them to be migrated to another server.
- ***Performance***: Conventional in-kernel packet processing is now much slower than newer options such as DPDK.

A tightly coupled user/kernel split is the root of these problems, so to solve them OVS must embrace a new architecture. Sections 2.2.1 and 2.2.2 describe two strategies that we have explored, but not adopted, because they give up compatibility with existing tools or perform poorly.

We finally converged on a third approach, which is presented in this paper. The main idea is to bring most of the packet processing into userspace using a new Linux socket type called AF_XDP [13]. We discuss how this solution guarantees compatibility with existing software and report on the challenges we faced to enable high performance packet processing (Section 3). We then describe how we integrated the new OVS with our network virtualization solution (Section 4), i.e., VMware NSX [77], and evaluated it using production-grade settings (Section 5) on different communication scenarios, i.e., intra-host and inter-host VM to VM as well as container to container. The new OVS approaches or matches the performance of OVS with DPDK: for container networking, it even beats DPDK processing latency by 12×, with 4× the throughput. Besides the pure performance capabilities, the benefits introduced by the new OVS have been substantial. With this new architecture, OVS can be now easily installed and upgraded in customer virtualized deployments. Moreover, validation and troubleshooting have been simplified. We detail these and other lessons learned (Section 6).

This paper makes the following contributions:

- We make the case for adopting AF_XDP as packet I/O channel for software switches, presenting a design of a userspace datapath for OVS.
- We demonstrate the implementation and performance optimizations of AF_XDP userspace driver for OVS.
- We show how the new OVS integrates with existing network virtualization solutions, such as NSX.
- We evaluate the new OVS using production-grade configurations.
- We discuss the challenges alongside associated solutions and the lessons we learned after two years spent in trying to migrate the datapath of OVS from in-kernel to userspace.
- We merged our code into the OVS upstream repository [26, 73]. Documentation and artifact evaluation information can be found here: https://github.com/williamtu/sigcomm21-ovs-artifacts

This work does not raise any ethical issues.

2 ENTERPRISE CHALLENGES

VMware NSX is an enterprise product, which means that customers install it in their own physical and virtual data center environments. Open vSwitch, the NSX dataplane on Linux hypervisors and in public clouds, must therefore support customer deployment scenarios ranging across many Linux distributions, kernel versions and variants, and physical and virtual hardware. The following sections explore the consequences.

2.1 Maintainability

OVS is a separate project from Linux, with different goals and different developers. To add a new OVS feature with a kernel component, OVS developers must convince the Linux kernel team to accept it. Linux kernel developers expect the new code to be well motivated, easily understood, necessary for performance, and maintainable over time. Furthermore, kernel policy forbids changing behavior that a user program might rely on [65]. This policy benefits users because programs that work with a particular Linux kernel will still work with any later one. The policy includes Open vSwitch userspace, that is, if kernel version x supports a given version of OVS, then so must every kernel version $y > x$. Thus, the OVS kernel module cannot change, but only extend, the interface it offers to userspace. The kernel module has hundreds of lines of advice on how to extend it correctly [55], but the level of difficulty is high enough that developers are still reluctant to do so.

The high bar to acceptance discourages developers from proposing features that would require experimentation or refinement, and Linux developers do reject some new OVS features. For example, they rejected support for the Stateless Transport Tunneling (STT) encapsulation due to claimed effects on routers [46, 60] and for an exact-match flow cache because of an objection to caches as a networking software design principle [47, 61].

For other features that do see eventual acceptance, the tight user/kernel coupling causes years of delay. Such features come along frequently, e.g. due to new protocols being standardized, such as VXLAN, NVGRE, Geneve, and NSH in past years. After code review, a new feature is merged to the next upstream Linux kernel version. Linux distribution vendors such as Ubuntu and Red Hat then infrequently update their kernels from upstream. Finally, data center operators often wait a considerable time after release to update their distributions. Therefore, new OVS features with kernel components become available to production users one to two years, or even longer, after the kernel component is merged.

> *Takeaway #1:* A split user/kernel design limits and delays feature support in software switches.

2.1.1 Out-of-Tree Kernel Module. In the past, OVS developers partially addressed the problem of slow feature delivery by supplying sources for an up-to-date OVS kernel module with OVS. Whereas a given Linux kernel release, say 4.11 or 5.9, does not receive new features over time, only bug fixes, the kernel module supplied with OVS, often called the "out-of-tree" kernel module, includes all the latest features and works with a wide range of kernel versions. In practice, the out-of-tree kernel module enables the latest features without upgrading the kernel.

The out-of-tree kernel module is, however, an evergreen source of problems. As a matter of policy, new features always start out in the upstream Linux tree, since going the other direction risks adding a feature to OVS that will never be accepted upstream. Thus, the out-of-tree module is always in the process of catching up to the

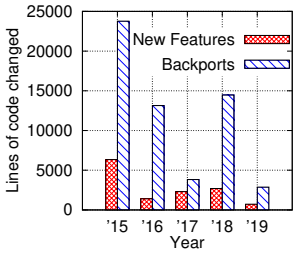

Figure 1: Lines of code changed in the last 5 years in the OVS repository's kernel datapath.

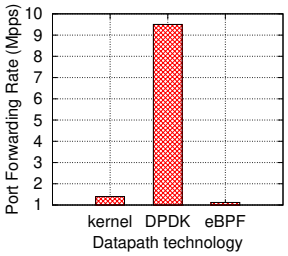

Figure 2: OVS forwarding performance for 64-byte packets and a single core.

upstream one. Figure 1 shows the lines of code added or removed in the out-of-tree module to add new upstream features under the label "New Features."

Second, the out-of-tree module requires developers to run faster and faster just to stay in the same place. The "Backports" bars in Figure 1 shows that the module requires thousands of lines of code changes every year just to stay compatible with new kernel releases. This can be more work than actually adding desired features. For example, consider our contribution of ERSPAN support to Open vSwitch, which added about 50 lines of code in the kernel module [68, 69]. Older kernels didn't have the IPv6 GRE infrastructure that this patch built on, so we had to add over 5,000 lines that combined 25 upstream commits to the out-of-tree module to support it there [57]. We still had to follow up to fix bugs, both those discovered in Linux and those we introduced in porting. Another example is per-zone connection limiting. In the kernel, we added about 600 lines for the feature [79]. In the out-of-tree module, we also had to add 700 lines based on 14 upstream commits, followed later by a combination of 14 more for bug fixes. Each commit we looked at for these features took considerable engineer effort to understand, compare among kernel versions, update, and review.

Third, one must validate an out-of-tree kernel module on every deployment target. Considering just NSX customers, this is a multitude considering the kernel versions used across cloud providers, common enterprise Linux distributions such as SLES or RHEL, and less well-known Linux distributions such as IBM LinuxONE and Oracle Linux. Also, some of these distributions require complicated steps to load out-of-tree modules.

Fourth, enterprise distributions will not support systems with third-party kernel modules. Production users often have support contracts with distributors, so adding new features in the out-of-tree module is not a viable way to support these users.

For these reasons, developers have judged the costs of maintaining the OVS out-of-tree kernel module to exceed the benefits. OVS 2.15, released in February 2021, deprecated the out-of-tree module and froze the latest supported kernel version. OVS 2.18, to be released in August 2022, will remove it entirely [58].

> *Takeaway #2:* An "out-of-tree" kernel module does not fix problems caused by a split user/kernel software switch design.

Command	Purpose
$ **ip link**	network device configuration
$ **ip address**	display and modify IP addresses
$ **ip route**	display and alter the routing table
$ **ip neigh**	display and alter the ARP table
$ **ping**	check L3 connectivity
$ **arping**	check L2 connectivity
$ **nstat**	display network statistics
$ **tcpdump**	packet analysis

Table 1: Commands commonly used to configure and troubleshoot Linux kernel managed network devices. These commands do not work on a NIC managed by DPDK.

2.2 Architecture

Once we accept that OVS's traditional architecture, with a userspace process tightly coupled to a kernel component, is the root of the problems that we highlighted in the foregoing sections, we can start to consider alternatives. We did not consider adopting other software switches, such as BESS [30] or VPP [4], because they do not support OpenFlow or the OVS extensions that NSX uses. Enabling NSX on them would have entailed a significant engineering effort on software stacks new to us. The obvious solution was to start from OVS and rethink its architecture.

2.2.1 All-Userspace OVS with DPDK. We can solve the architectural problem by moving all of OVS to userspace. For years, OVS has supported all-userspace software switching using DPDK [17, 53], a library to accelerate packet processing workloads. This avoids the maintainability challenges that come from spanning Linux and OVS development. Additionally, OVS with DPDK can simply be restarted for upgrades or bug fixes as no kernel upgrade or reboot is needed. Using DPDK frees OVS from maintaining compatibility with rapidly changing Linux internal interfaces, but DPDK also changes the interfaces it presents to software like OVS from one version to the next. The corresponding changes to OVS are significant enough that each OVS release requires a specific version of DPDK and that OVS needs a dedicated branch in its repository to keep up with DPDK compatibility.

OVS with DPDK suffers from a different class of compatibility problems, which have kept it from the success of its in-kernel counterpart [81]. DPDK uses its own network device drivers and operates without kernel involvement, so well-known tools to configure, manage, and monitor NICs, including those listed in Table 1, do not work with NICs in use by DPDK, which makes DPDK setups difficult to manage and debug [32]. Instead, users have to use DPDK-specific tools, i.e, `testpmd` [18], `dpdk-pdump`, `dpdk-procinfo` [19]. This is a problem in large-scale deployments that include VMs, containers, physical machines over a variety of Linux distributions and team collaborations, because the additional maintenance of the tools and operational cost make this approach unattractive.

Administrators must dedicate NICs to use with DPDK, which means that they must maintain two separate networking configurations, one for the kernel, one for DPDK, increasing their management burden. DPDK is also notorious for its strict system requirements and unstable API [43]. Finally, DPDK performance requires dedicating one or more CPU cores to switching, a choice that non-NFV customers usually reject in public cloud deployments because of per-core pricing [76].

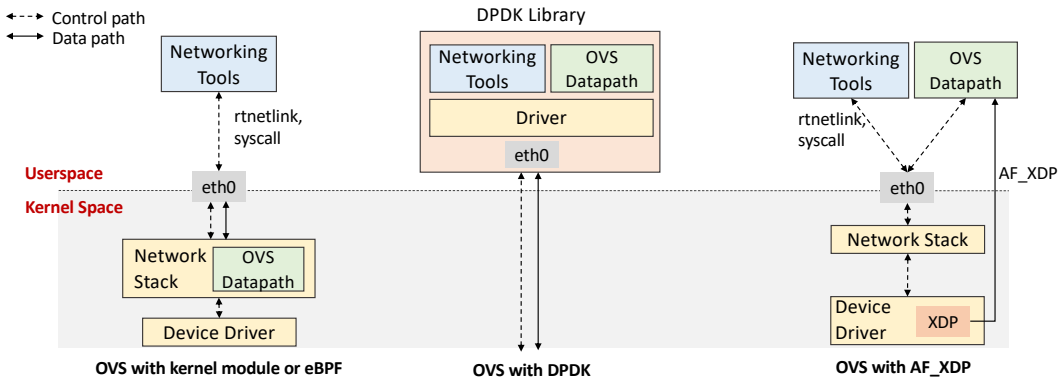

Figure 3: Comparison between standard OVS, OVS attached to a DPDK device and OVS attached to a Linux kernel managed device with AF_XDP.

> *Takeaway #3:* DPDK is fast, simplifies upgrade, and eases work for developers, but it is incompatible with the tools and systems that users expect to use.

2.2.2 Decoupling the Kernel with eBPF. It is also possible to retain OVS's existing user/kernel division, but replace the kernel module by an eBPF program. eBPF [25] allows specially written programs to safely run in an in-kernel sandbox, without changing kernel source code or loading kernel modules. Distributions are willing to support third-party eBPF programs because of eBPF's safe, sandboxed implementation. An implementation of OVS based on eBPF would solve maintainability and operability problems and retain compatibility with customary tools.

eBPF, however, has substantial limitations. The sandbox limits the size of an eBPF program, and though it lets multiple programs be chained together, it still restricts OVS capabilities. The sandbox also caps eBPF complexity by disallowing loops. These restrictions mean that the eBPF datapath lacks some OVS datapath features. The sandbox restrictions, for example, preclude implementing the OVS "megaflow cache" that is important for performance in many situations.[1]

Performance is also important. A sandboxed, bytecode implementation of code in eBPF runs slower than comparable C. Figure 2 compares the performance of OVS in practice across three datapaths: the OVS kernel module, an eBPF implementation, and DPDK. The test case is a single flow of 64-byte UDP packets, which is a case that does not rely on the megaflow cache for performance. Unfortunately, the sandbox overhead makes eBPF packet switching 10–20% *slower* than with the conventional OVS kernel module [66]. This is enough to disqualify it from further consideration.

> *Takeaway #4:* eBPF solves maintainability issues but it is too slow for packet switching.

2.2.3 Mostly Userspace OVS with AF_XDP. Finally, we settled on a third approach using AF_XDP, a socket address family available in Linux 4.18 and later [13]. AF_XDP builds on XDP [8, 32, 48], which is a "hook point" in each NIC network driver where a userspace program may install an eBPF program. The NIC driver executes the XDP program on every received packet, even before it takes the expensive step of populating it into a kernel socket buffer data structure.

AF_XDP gives an XDP hook program the ability to send its packet to userspace across a high-speed channel (see Section 3), bypassing the rest of the Linux networking stack. To use AF_XDP, OVS installs an XDP hook program that simply sends every packet to OVS in userspace.

OVS with AF_XDP reduces the coupling between OVS and the kernel to that of a tiny eBPF helper program which is under the Open vSwitch community's control. The helper program just sends every packet to userspace, in contrast to the eBPF program described in the previous section, which had the same complexity and feature set as the OVS kernel module. In addition, XDP and AF_XDP are "stable" kernel interfaces: kernel developers commit that XDP programs and AF_XDP sockets will continue to work the same way they do now in future kernel versions. OVS implements its own AF_XDP driver, even though OVS could use DPDK's AF_XDP Poll Mode driver [16]. Initially, this was because the OVS AF_XDP driver predated the one in DPDK. Now, for non-performance critical customers, deploying OVS with DPDK just to use its AF_XDP driver consumes too much CPU and memory. Using the DPDK driver would also force the OVS community to keep up with changes in DPDK, increasing maintenance overhead.

Using AF_XDP also simplifies upgrades and bug fixes from rebooting or reloading a kernel module to simply restarting OVS. Most importantly, AF_XDP provides compatibility with the same Linux networking tools as in-kernel OVS and with every Linux distribution.

Figure 3 conceptually compares the standard OVS with in-kernel datapath, OVS in eBPF, with DPDK and with AF_XDP. The first two are merged together because in both cases the OVS dataplane resides in the Linux network stack: while the standard OVS datapah is a kernel module, the other is a eBPF program attached to the *traffic control (tc)* hook of the kernel. With DPDK, a userspace driver for the NIC entirely bypasses the Linux kernel, which creates the

[1]The megaflow cache could be implemented as a new form of eBPF "map" but the kernel maintainers rejected this proposal [71].

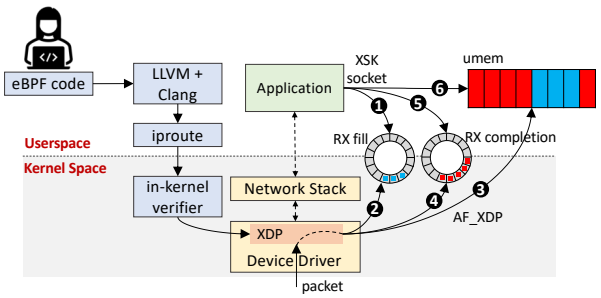

Figure 4: The workflow of XDP eBPF development process and the AF_XDP packet flow on the receiving side.

compatibility issues discussed above. With AF_XDP, OVS and Linux use the standard kernel NIC driver, which supports the standard system calls and rtnetlink interface [49, 74], and that in turn means that the existing monitoring and management tools based on those continue to work. Later, we will also discuss some cases where a small amount of additional sophistication in the XDP hook program provides additional benefits.

3 OVS WITH AF_XDP

This section provides some background on AF_XDP and discusses the challenges we faced in implementing OVS with this new Linux socket type. The main objective was to optimize our system so to provide performance comparable to kernel-bypass solutions while retaining compatibility with existing tools for network monitoring and management.

3.1 Background on AF_XDP

AF_XDP gives an XDP hook program the ability to send its packet to userspace, bypassing the rest of the Linux networking stack. On the left side of Figure 4, we show the typical workflow for installing an eBPF program to an XDP hook point. Clang and LLVM take a restricted C program as input and output it to an ELF object file, which contains the eBPF instructions. An eBPF loader, such as `iproute`, then issues appropriate BPF syscalls to verify and load the program. If the program passes this stage, it is attached to the hook point (in this case XDP). Subsequently, any packet that reaches the XDP ingress hook will execute the eBPF program. The XDP program in the figure sends every received packet to userspace via the AF_XDP socket or "XSK", which is the user/kernel interface to this mechanism. Each XSK uses an Rx and a Tx data structure in userspace, each of which in turn points to a memory area called a *umem*. A umem consists of two rings: a *fill ring* and a *completion ring*. To receive packets, the userspace program appends empty descriptors to the Rx fill ring (path ❶). When a packet arrives, the kernel pops a descriptor from the ring (path ❷), writes packet data into its attached memory (path ❸), and pushes it onto the Rx completion ring (path ❹). Later, the userspace program fetches packet data from descriptors on the Rx completion ring (path ❺) so it knows where to find the packets to process in umem (path ❻). After it finishes processing them, it re-attaches them to the Rx fill ring to receive new incoming packets. Sending packets works in a similar way using the Tx fill and completion rings.

The following subsections describe our work for physical devices, virtual devices, and containers. The performance results quoted in these sections are from back-to-back Xeon E5 2620 v3 12-core (2.4 GHz) servers, each with a Mellanox ConnectX-6 25GbE NIC and running Ubuntu kernel 5.3. One server generated a single UDP flow with 64-byte packets, which a second server processed using OVS with AF_XDP. We report the maximum throughput sustained by OVS before starting to drop packets.

Optimizations	Rate (Mpps)
none	0.8
O1	4.8
O1+O2	6.0
O1+O2+O3	6.3
O1+O2+O3+O4	6.6
O1+O2+O3+O4+O5	7.1*

Table 2: Single-flow packet rates for 64-byte UDP between a physical NIC and OVS userspace with different optimizations. *Estimated

3.2 Physical Devices

Table 2 shows the performance we obtained for our physical NICs along with the various optimizations we implemented. The following subsections provide further details.

O1: Dedicated thread per queue. By default, the OVS "userspace datapath" processes packets with the same general-purpose thread that it uses for other tasks, including OpenFlow and OVSDB processing. This is the same code and the same approach that OVS used with DPDK, by default. However, when we used `strace` to analyze the pattern of system calls in this model, we found that it spent a lot of CPU time polling. We addressed the problem by configuring PMD (poll-mode-driver) threads, which are threads dedicated to packet processing. Each PMD thread runs in a loop and processes packets for one AF_XDP receive queue. Enabling PMD threads improved performance by 6×, from 0.8 Mpps to 4.8 Mpps.

O2: Spinlock instead of mutex. OVS configures a region of userspace memory so that, with kernel assistance, network devices can directly access it. We wrote a userspace library called `umempool` to manage this memory, as in [72]. The umem regions require synchronization, even if only one thread processes packets received in a given region, because any thread might need to send a packet to any umem region. We initially used a POSIX mutex to synchronize access. Linux `perf` showed that the threads spent around 5% of their CPU time in the mutex function `pthread_mutex_lock`, even when we deployed a single queue with a single thread. We realized later that locking a mutex could context switch the current process. Thus, we switched to spinlocks, which have less than 1% overhead when there is no contention. This improved performance by 1.25×, from 4.8 Mpps to 6.0 Mpps.

O3: Spinlock batching. The basic AF_XDP design assumes that packets arrive in a userspace rx ring in batches. This allows OVS to fetch multiple packets at the same time. However, each batch receive operation requires some additional housekeeping work. For example, each successful batch receive is also paired with a request from the umempool for more usable buffers to refill the XSK fill ring for the next incoming packets. Each operation needs a lock to prevent contention. Linux `perf` showed that our initial lock

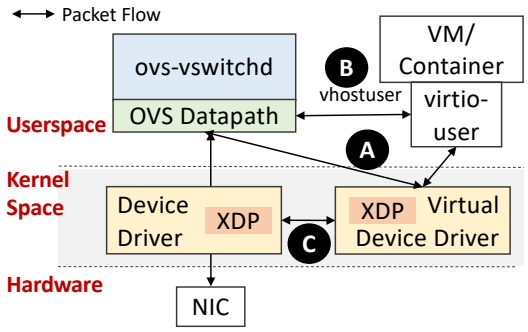

Figure 5: A physical NIC with AF_XDP support can directly DMA packet buffers into the OVS userspace datapath. This might not be helpful in the presence of virtual devices.

implementation, which gave each packet in each umempool its own spinlock, was too fine-grained. Therefore, we optimized it with a more coarse-grained lock that batched multiple packets together. We also batched multiple umempool accesses, and shared multiple operations across a spinlock. This improved the performance by 1.05×, from 6.0 Mpps to 6.3 Mpps.

O4: Metadata pre-allocation. OVS uses the `dp_packet` structure to track each packet's metadata, including its input port, the offset of its L3 header, and a NIC-supplied hash of the packet's headers. Using `strace`, we noticed that the `mmap` system call used to allocate `dp_packet` structures entailed significant overhead. To avoid it, we pre-allocated packet metadata in a contiguous array and pre-initialized their packet-independent fields. Using a single array also improved cache locality. This optimization increased performance by 1.04×, from 6.3 Mpps to 6.6 Mpps.

O5: Checksum offload. Modern NICs offer hardware features to reduce the CPU cost of packet processing, such as checksum calculations, pushing and popping VLAN tags, and TCP segmentation. Both the Linux kernel and DPDK support these offloads in their hardware NIC drivers. AF_XDP does not yet, although it has a framework to support them through a generic NIC metadata area defined using a format called BTF (BPF Type Format) and XDP hints [39, 78]. We have been working with the XDP community and device vendors to get checksum offload support from their drivers. To estimate the potential for improvement from AF_XDP checksum offload, we modified our code to transmit a fixed value for the packet's checksum and to assume at receive time that the checksum is correct. This improved performance by 1.07×, from 6.6 Mpps to 7.1 Mpps. We expect better results with bigger packets because the checksum's cost is proportional to the packet's payload size. TCP segmentation offload can also offer major benefits for bulk TCP workloads but it is not relevant for the below-MTU size packets that we used here.

3.3 Virtual Devices

The optimizations discussed above enable fast packet I/O between a physical NIC and OVS userspace, but not between OVS userspace and a VM or container within the same host.

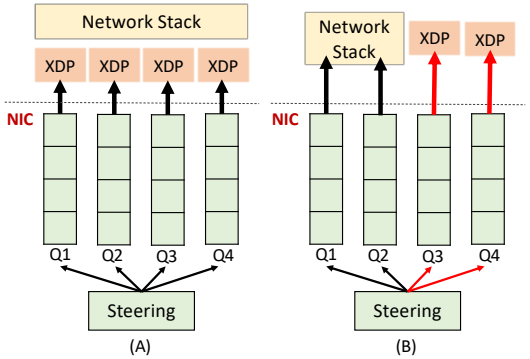

Figure 6: XDP program can be attached to either all or a subset of the receive queues, depending on the vendor's driver.

To send packets to a local VM, OVS with AF_XDP typically uses the `sendto` system call to send packets through a Linux kernel virtual "tap" interface, following path A in Figure 5. We measured the cost of this system call as 2 μs on average, which is much more than in the traditional OVS user/kernel model, where passing a packet to the tap device was just an intra-kernel function call with the data already resident in kernel memory. This drops the 7.1 Mpps throughput achieved by the previous optimizations to just 1.3 Mpps. To overcome this issue, we reconfigured our VM to directly use the same vhost protocol that the kernel's tap device implements [3, 29]. Using this "vhostuser" implementation, packets traverse path B, avoiding a hop through the kernel. This change increased performance to 6.0 Mpps, similar to a physical device.

3.4 Containers

Most non-NFV applications rely on kernel sockets for network access, which means that regardless of how packets arrive at a VM or container, a kernel TCP/IP stack must initially receive and process them. For a VM, the kernel in question is the VM's kernel; for a container, it is the host kernel. For two containers within the same host, the in-kernel switch works efficiently across a kernel "veth" virtual device, which passes packets from one kernel network namespace to another without a data copy. In this case, adding a round trip through OVS userspace, via AF_XDP or otherwise, hurts performance due to extra context switches and packet copies.

We are still exploring ways to optimize this case. Enabling zero-copy AF_XDP in veth [67] would eliminate the packet copy penalty. Another option is to modify the OVS XDP program to bypass OVS userspace for the container's packets, sending them directly to its veth driver, along path C. This solution avoids expensive copies of the data from kernel to userspace, at the cost of adding logic in the XDP program. Section 5.4 will explore performance trade-offs to adding logic at the XDP level. Finally, if the container application can be modified, users can use dpdk-vhostuser library for container applications [20, 62] and a userspace TCP/IP stack [23, 28].

3.5 Extending OVS with eBPF

Our new model for OVS attaches a simple eBPF program to the XDP hook point to send packets to OVS in userspace through an AF_XDP socket. One may extend this program to implement new functionality at the device driver level. One example is what discussed above in the context of container networking. Another

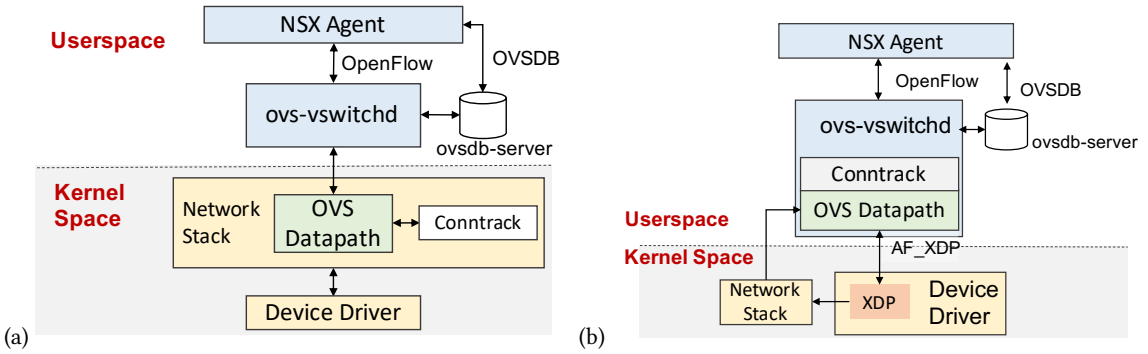

Figure 7: Our network virtualization system, NSX, integrated with: (a) the OVS kernel datapath and (b) OVS with AF_XDP.

example is to implement an L4 load-balancer in XDP targeting a particular 5-tuple, which directly processes any packet that matches the 5-tuple and passes non-matching packets to the userspace OVS datapath. These examples benefit from avoiding the latency of extra hops between userspace and the kernel. Other reasons to extend the eBPF program include integrating third-party eBPF programs with Open vSwitch and dividing responsibility for packet processing across multiple userspace daemons with multiple AF_XDP sockets.

eBPF might be especially useful for integrating with 3rd-party software that is not amenable to implementation via the OpenFlow protocol that OVS supports. Many high-level languages support eBPF bytecode as their compile target. Developers can write in C and compile with LLVM and Clang, or use another language with an eBPF backend, such as P4 [9, 70], Rust [22], and Python [7]. Furthermore, XDP has a well-defined runtime interface to interact with the kernel, allowing third-party XDP programs to easily integrate into OVS and to be updated without restarting OVS.

Limitations. Intel, Mellanox, and other vendors have already added Linux driver support for AF_XDP. With NICs that still lack full support, OVS uses a fallback mode that works universally at the cost of an extra packet copy, reducing performance [15]. Moreover, vendors support different models for attaching XDP programs to a network device.

Figure 6(a) shows the attachment model currently adopted for Intel NICs, in which all traffic coming into the device triggers the XDP program. Any program that needs to distinguish between different types of traffic, e.g. to treat management traffic differently, must implement XDP program logic to parse the protocol and forward the desired traffic to the Linux network stack.

Figure 6(b) shows the attachment model currently adopted for Mellanox NICs, in which an XDP program can be attached to a subset of a device's receive queues, e.g. only queues 3 and 4 as shown in the figure. In this model, users may program the NIC hardware flow classification rules, e.g. with `ethtool --config-ntuple`, to distinguish different type of traffic in hardware, then write simpler per-queue programs. This model is more flexible, although hardware has limited matching fields and therefore more complicated case still require software steering logic.

4 INTEGRATING OVS WITH NSX

NSX is a network virtualization system that, in 2019, VMware reported to be used by 10,000 customers, including 82 of the Fortune 100 [31]. NSX allows an administrator to overlay a virtual network with L2 and L3 elements, firewalling and NAT, and other features, on top of a physical network of hypervisors, bare-metal hosts, and public cloud VMs. NSX uses OVS as its dataplane for Linux hypervisors and public cloud VMs.

Figure 7(a) and (b) show how NSX integrates with OVS on a hypervisor in the existing and new model, respectively. In both models, the NSX controller has an agent that connects remotely to the NSX central controller using a proprietary protocol and locally to OVS using OVSDB and OpenFlow. The NSX agent uses OVSDB, a protocol for managing OpenFlow switches, to create two bridges: an *integration bridge* for connecting virtual interfaces among VMs, and an *underlay bridge* for tunnel endpoint and inter-host uplink traffic. Then it transforms the NSX network policies into flow rules and uses the OpenFlow protocol to install them into the bridges. The agent uses both protocols to maintain and monitor bridge, device, and flow state. The NSX agent can also configure OVS to integrate with third-party packet processing software, such as Deep Packet Inspection (DPI) engines.

Figure 7(a) shows NSX using OVS in the traditional split user/kernel model. The OVS userspace `ovs-vswitchd` process works with the datapath kernel module, which in turn interfaces to other parts of the kernel, including the TCP/IP network stack, the connection tracking module, and device drivers.

Figure 7(b) shows NSX using OVS with AF_XDP. Here, the packet processing datapath is outside the kernel, inside userspace `ovs-vswitchd`. OVS manages the XDP program: it uses the kernel version to determine the available XDP features, detects the LLVM and Clang version for compiling the eBPF program, and loads and unloads XDP programs when users add or remove a port in the OVS bridge.

By moving packet processing to userspace, bypassing the Linux networking subsystem entirely, OVS loses access to many services provided by the kernel's networking stack. NSX depends on many of these, such as connection tracking for firewalling in the kernel's netfilter subsystem, the kernel's GRE, ERSPAN, VXLAN, and Geneve encapsulations, and the quality of service (QoS) traffic policing and shaping features from the kernel's TC subsystem. Therefore,

Entity	Count
Geneve tunnels	291
VMs (two interfaces per VM)	15
OpenFlow rules	103,302
OpenFlow tables	40
matching fields among all rules	31

Table 3: Properties of the OpenFlow rule set, taken from one of our hypervisors, used for performance evaluation.

OVS uses its own userspace implementations of these features, built by OVS developers over a period of years [52]. NSX accesses these features via OVSDB and OpenFlow, not directly through the kernel, so it needs no changes to use the userspace implementations.

To implement L3 encapsulations such as Geneve, OVS must support IP routing and ARP (for IPv4) or ND (for IPv6). OVS implements all of these in userspace too, but the NSX agent and other hypervisor software directly use the kernel's implementations of these features. Therefore, to maintain compatibility, OVS caches a userspace replica of each kernel table using Netlink, a Linux-specific socket type for configuring and monitoring the kernel. Using kernel facilities for this purpose does not cause performance problems because these tables are only updated by slow control plane operations.

OVS also needs a reliable TCP/IP stack to allow the switch to connect to its controller. We could use a userspace TCP/IP stack [23, 28], but we were not convinced they are production ready. Instead, we decided to use the Linux kernel TCP/IP stack by setting up a tap device for injecting packets from userspace into the kernel and vice versa. This is slow, but it is acceptable because management traffic does not require high throughput. If it proves too slow later, we can modify the XDP program to steer the control plane traffic directly from XDP to the network stack, while keep pushing dataplane traffic directly to userspace with the AF_XDP socket.

5 PERFORMANCE EVALUATION

In this section we evaluate the performance of OVS version 2.14 with AF_XDP and compare it against its in-kernel and DPDK 20.05 counterparts in different scenarios.

5.1 NSX Performance

Our first test aims at reproducing a full NSX deployment. We used two Intel Xeon E5 2440 v2 2.4 GHz servers connected back-to-back, both with 8 cores and hyperthreading enabled, running Ubuntu Linux with kernel 5.3.0 and equipped with Intel 10 GbE X540 dual-port NICs. Both servers ran OVS and an NSX agent deploying around 103,000 production grade OpenFlow rules taken from one of our hypervisors, to mimic a real world deployment.

Table 3 shows some statistics about the NSX OpenFlow flow tables, which include Geneve tunneling and a distributed firewall with connection tracking. With these flow tables, many packets recirculate through the datapath twice. The initial lookup examines the packet's outer Ethernet header, determines that it is a tunneled packet that NSX must handle, and directs it to the NSX "integration bridge". There, the second lookup matches the inner packet, figures out that firewall connection tracking is needed and the appropriate "zone" to ensure separation between different virtual networks, and passes the packet and zone to the connection tracking module. The third lookup incorporates the connection tracking state and chooses the appropriate forwarding action.

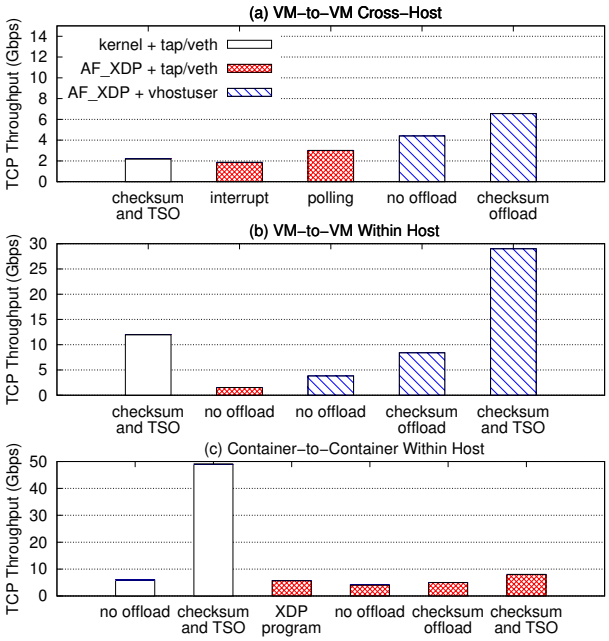

Figure 8: TCP throughput in three different scenarios, with patterns for datapaths and virtual device types, and labels for optimizations applied.

We evaluated three scenarios: inter-host VM-to-VM, intra-host VM-to-VM, and intra-host container-to-container. In each scenario, we tested OVS with its traditional split user/kernel datapath and with AF_XDP. We did not test with DPDK because operators are unlikely to use it in real NSX deployments (see Section 2.2.1). In each test, we used `iperf` to send a single flow of bulk TCP packets. Each packet passed through the OVS datapath pipeline three times, as described above, which makes it a useful test of the cost of recirculation and the impact of optimizations such as TSO (TCP Segmentation Offloading).

Figure 8(a) shows VM-to-VM performance across a pair of hosts, using Geneve encapsulation across a 10 Gbps link. The first bar shows that the kernel datapath achieves 2.2 Gbps of throughput using tap devices. The second and third bars reflect a switch to AF_XDP, still with a tap device. The second bar shows that using AF_XDP in an interrupt-driven fashion, which cannot take advantage of any of the optimizations described in Section 3, yields only 1.9 Gbps of performance. When we switch to polling mode and apply optimizations O1…O4, the third bar shows that throughput rises to about 3 Gbps. The remaining bars show the additional benefits of switching to a vhostuser device: 4.4 Gbps without checksum offload, 6.5 Gbps with checksum offload. We estimated the benefit of offloading, as described in Section 3.

Figure 8(b) shows VM-to-VM performance within a single host. The first bar shows about 12 Gbps of throughput using tap devices with the standard kernel datapath. This is mainly a consequence of TSO, which allows the kernel to handle 64-kB packets, and checksum offloading (within a single host, this means not generating a checksum at all). The second bar switches to AF_XDP with a tap device, and the remaining bars are for AF_XDP with vhostuser

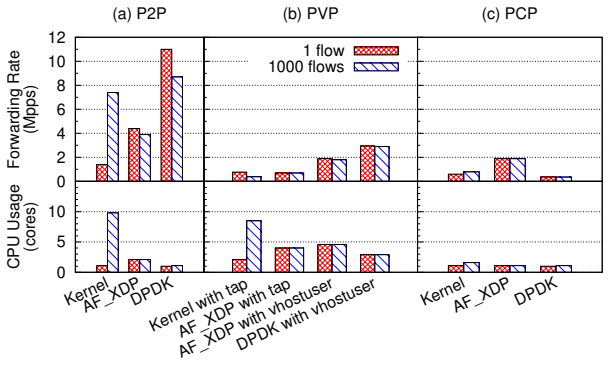

Figure 9: The forwarding rates and CPU consumption of Physical-to-Physical (P2P), Physical-to-Virtual-to-Physical (PVP) and Physical-to-Container-to-Physical (PCP) communication across datapaths and I/O drivers.

devices, which consistently perform better than tap, because vhostuser avoids context switching between the kernel and userspace (as discussed in Section 3.3). The three vhostuser bars show the value of offloading: without offloads, throughput is 3.8 Gbps; offloading checksum improves performance to 8.4 Gbps; and enabling TSO achieves 29 Gbps. The final configuration outperforms the kernel datapath because vhostuser packets do not traverse the userspace QEMU process to the kernel.

Finally, Figure 8(c) shows container-to-container performance within a host. All of these use veth virtual interfaces between namespaces, as described in Section 3.4. The first two bars in this graph show that the kernel achieves about 5.9 Gbps throughput with TSO and checksum offloading disabled and that this grows to 49 Gbps when they are enabled. The third bar shows about 5.7 Gbps throughput when we use the new OVS leveraging XDP redirection [41], following path C in Figure 5. This is slower than in-kernel OVS because XDP does not yet support checksum offload and TSO [40]. The last three bars show performance if, instead, we send all the packets to the userspace OVS via AF_XDP then forward them to the appropriate destination veth, following path A in Figure 5: 4.1 Gbps without offloading, 5.0 Gbps with checksum offloading, 8.0 Gbps with checksum offloading and TSO.

Outcome #1. For VMs, OVS AF_XDP outperforms in-kernel OVS by 2.4× within a host, rising to about 3× across hosts. For container networking, however, in-kernel networking OVS remains faster than AF_XDP for TCP workloads for now. (We expect upcoming support for TSO in AF_XDP to reduce or eliminate this disadvantage.)

5.2 Packet Forwarding Rate

The previous section measured performance of an imitation production deployment that included firewalling, tunneling, and other features. This section instead perform a raw packet forwarding microbenchmark, measuring the packet forwarding rate and CPU usage. For these tests we switched to a testbed with two servers (Xeon E5 2620 v3 12-core 2.4GHz) connected back-to-back through dual-port 25-Gbps Mellanox Connect-X 6Dx NICs. One server ran the TRex [11] traffic generator, the other ran OVS with different datapaths and packet I/O configurations as well as a VM with 2 vCPUs and 4 GB memory. We tested three scenarios, all loopback configurations in which a server receives packets from TRex on one NIC port, forwards them internally across a scenario-specific path, and then sends them back to it on the other. We measured OVS performance with the in-kernel datapath, with AF_XDP, and with DPDK.

In each case, we measured the maximum lossless packet rate and the corresponding CPU utilization with minimum-length 64-byte packets, for 1 flow and 1,000 flows. With 1,000 flows, we assigned each packet random source and destination IPs out of 1,000 possibilities, which is a worst case scenario for the OVS datapath because it causes a high miss rate in the OVS caching layer for the OpenFlow rules specified by the control plane [56].

Figure 9(a) shows the results in a physical-to-physical (P2P) scenario, in which OVS forwards packets directly from one physical port to another, to measure packet I/O performance with minimum overhead. For all of the userspace datapath cases, 1,000 flows perform worse than a single flow because of the increased flow lookup overhead. The opposite is true only for the kernel datapath because of kernel support for receive-side scaling (RSS), which spreads the flows across multiple CPUs. It is fast, but not efficient: the kernel uses almost 8 CPU cores to achieve this performance. DPDK outperforms AF_XDP in terms of forwarding rate capabilities while keeping the CPU utilization quite low.

Figure 9(b) shows the results in a physical-virtual-physical (PVP) scenario, which adds a round trip through a VM to each packet's path. For VM connectivity, we tested the kernel with a tap device, DPDK with vhostuser, and AF_XDP with both possibilities. We see that vhostuser is always better than tap, with higher packet rate and lower CPU utilization (explained in Section 3.3). With vhostuser, AF_XDP is about 30% slower than DPDK. CPU usage is fixed regardless of the number of flows across all the userspace options, because all of these cases use preconfigured CPU affinity. The kernel datapath, on the other hand, uses the RSS feature of NIC hardware to balance packet processing across all 16 hyperthreads. In other scenarios this can boost performance, but here it does not because the kernel does not support packet batching, busy rx/tx ring polling, batched memory pre-allocation, or other optimizations incorporated into the OVS userspace datapath [12].

Figure 9(c) reports the physical-container-physical (PCP) scenario, which adds a round trip through a container to each packet's path. For container connectivity, we tested the kernel with a veth device and AF_XDP and DPDK with vhostuser. Here, AF_XDP performs best, both in speed and CPU use, because it processes packets in-kernel using an XDP program between the physical NIC and the container, following path C in Figure 5. Compared to DPDK, it avoids the costly userspace-to-kernel DPDK overhead; compared to the kernel, the XDP program avoids much of the kernel's ordinary overhead.

Table 4 details the 1,000-flow CPU statistics from Figure 9. Each column reports CPU time in units of a CPU hyperthread. The **system** and **softirq** columns are host kernel time, with **system** corresponding to system calls and **softirq** to packet processing; **guest** is all time running inside a VM (blank outside of PVP, the only scenario with a VM); **user** is time spent in host userspace (including

path	configuration	system	softirq	guest	user	total
P2P	kernel	0.1	9.7		0.1	9.9
	DPDK	0.0	0.0		1.0	1.0
	AF_XDP	0.1	1.1		0.9	2.1
PVP	kernel	1.2	6.0	1.1	0.2	8.5
	DPDK + vhost	0.9	0.0	1.0	1.0	2.9
	AF_XDP + vhost	0.9	0.8	1.0	1.9	4.6
PCP	kernel	0.0	1.5		0.0	1.0
	DPDK	0.3	0.5		0.2	1.0
	AF_XDP	0.0	1.0		0.0	1.0

Table 4: Detailed CPU use with 1,000 flows, in units of a CPU hyperthread. Red vertical bars are scaled with the numbers.

OVS userspace); and **total** sums the other columns. In the P2P and PVP scenarios, the kernel configuration mostly uses kernel CPU time and the DPDK configuration mostly runs in userspace. The AF_XDP configuration sits in the middle, with part of its processing in the kernel's softirq context for processing the XDP program and the rest spent in OVS userspace. In return for the minimal extra cost of in-kernel processing, we obtain the flexibility of extending OVS features through loading XDP programs into kernel. PVP behavior is similar to P2P. For PCP performance, all the three configurations show similar usage, instead.

Outcome #2. OVS AF_XDP outperforms the other solutions when the endpoints are containers. In the other settings, DPDK provides better performance. However, with more optimizations being proposed to the kernel community, e.g., optimizations for socket polling, memory management, zero-copy [5, 13, 14, 33, 37, 63, 64], we expect that the extra softirq CPU consumption in Table 4 will drop and as a result, show better CPU utilization than DPDK.

5.3 Latency and Transaction Rate

We conducted two more experiments to understand our design's impact on latency, using the same testbed as the previous section. In each experiment, we ran netperf's TCP_RR test, which sends a single byte of data back and forth between a client and a server as quickly as possible and reports the latency distribution. We tested OVS in three different configurations:

- Kernel: OVS connecting to VMs with tap devices and to containers with veth.
- DPDK: OVS with DPDK's mlx5 driver, connecting to VMs with vhostuser and containers with the DPDK AF_PACKET driver.
- AF_XDP: OVS with AF_XDP, connecting to VMs with vhostuser and containers using vhostuser and containers with an XDP program.

The first experiment tested inter-host latency. On one host, we ran the netperf server; on the other, we ran the client in a VM with 2 virtual CPUs. The top half of Figure 10 shows the 50th, 90th, and 99th percentiles latencies reported by netperf. The bottom half shows the same performance data interpreted as transactions per second. The kernel shows the highest latency, with P50/90/99 latencies of 58/68/94 μs. At 36/38/45 μs, DPDK performs much better than the kernel because it always runs in polling mode, whereas the kernel adaptively switches between interrupt and polling modes. AF_XDP's 39/41/53 μs latency barely trails DPDK, mainly because

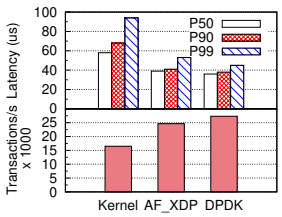

Figure 10: Latency and transaction rates between one host and a VM on another.

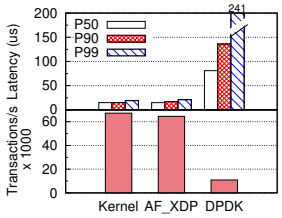

Figure 11: Latency and transaction rates between containers within a host.

AF_XDP lacks hardware checksum support, as mentioned in section 4.

The second experiment tested intra-host latency. On a single host, we ran the netperf server in one container and the client in another. Figure 11 reports the results. Here, the kernel and AF_XDP obtain similar results, with P50/90/99 latency about 15/16/20 μs, but DPDK is much slower at 81/136/241 μs. Section 3.4 explained the reason: packets to or from a container must pass through the host TCP/IP stack, which the OVS in-kernel and AF_XDP implementations can do cheaply but for which DPDK needs extra user/kernel transitions and packet data copies.

Outcome #3. OVS with AF_XDP performs about as well as the better of in-kernel or DPDK for virtual networking both across and within hosts.

5.4 The cost of XDP processing

Sections 5.1 and 5.2 presented the case where the userspace OVS datapath processes all the traffic received by the physical interface. This is because we used a minimal XDP program that we instructed to push all the data up to the stack. In this section, we reuse the testbed used in Section 5.1 to understand the cost of adding new features at the NIC driver level with XDP, by implementing certain flow processing logic in P4-generated XDP programs and measuring the performance using a single core.

Table 5 shows the results we obtained under different scenarios. Task A drops all incoming packets without examining them, which reaches 14 Mpps line rate for a 10 Gbps link. As the XDP program complexity increases, performance decreases. The basic L2/L3 parsing in task B drops the rate to 8.1 Mpps as the CPU now must read the packet (triggering cache misses) and branch through protocol headers. Task C adds an eBPF map table lookup and slows processing further, to 7.1 Mpps. Finally, in task D, which modifies and forwards the packet, the forwarding rate drops further to about 4.7 Mpps.

Outcome #4. Complexity in XDP code reduces performance. Processing packets in userspace with AF_XDP isn't always slower than processing in XDP.

5.5 Multi-Queue Scaling

Figure 12 shows the effect of using multiple NIC queues on OVS forwarding performance using the physical-to-physical configuration adopted in Section 5.2. One server ran the TRex traffic generator, the other ran OVS with DPDK or AF_XDP packet I/O with 1, 2, 4,

XDP Processing Task	Rate
A: Drop only.	14 Mpps
B: Parse Eth/IPv4 hdr and drop.	8.1 Mpps
C: Parse Eth/IPv4 hdr, lookup in L2 table, and drop.	7.1 Mpps
D: Parse Eth/IPv4 hdr, swap src/dst MAC, and fwd.	4.7 Mpps

Table 5: Single-core XDP processing rates for different tasks.

or 6 receive queues and an equal number of PMD threads. We generated streams of 64 and 1518 packets at 25 Gbps line rate, which is 33 Mpps and 2.1 Mpps, respectively, via back-to-back 25 Gbps Mellanox ConnectX-6Dx NICs. With 1518-byte packets, OVS AF_XDP coped with 25 Gbps line rate using 6 queues, while in the presence of 64-byte packets the performance topped out at around 12 Mpps, even with 6 queues. The DPDK version consistently outperformed AF_XDP in these cases.

We ran a few micro-benchmarks using Linux `perf` on the 6 PMD queues and we compared the results between OVS AF_XDP and DPDK to find out the reasons of such a difference in performance. With DPDK, the PMD threads spent most of their CPU cycles in userspace, processing packets. OVS AF_XDP had two major overheads: (1) it suffered from context switches into the kernel to transmit packets, and (2) it had to calculate *rxhash*, a hash value of the packet 5-tuple used for RSS (Receive Side Scaling) as there is no API for XDP to access the hardware offload features (yet [39]). Furthermore, the core utilization for DPDK is more efficient, as already mentioned in Table 4 P2P result.

Outcome #5. AF_XDP does not yet provide the performance of DPDK but it is mature enough to saturate 25 Gbps with large packets.

6 LESSONS LEARNED

We have spent two years working on migrating the OVS in-kernel datapath to the one presented in this paper. This section discusses the lessons we learned during this process.

✔ **Reduced risk.** A kernel module is a huge source of risk because any bug can crash the entire host and all the software running on it, whereas a bug in OVS with AF_XDP only crashes the OVS process, which automatically restarts. The reduction of risk has other benefits: users of AF_XDP will not lose enterprise Linux support from Red Hat or another vendor due to loading an out-of-tree kernel module, and production system admins might be more willing to deploy OVS and experiment with new versions and features.

✔ **Easier deployment.** Building and validating a kernel module across all the kernel versions where it is needed is difficult. With AF_XDP, the engineering team does not face this difficulty.

✔ **Easier upgrading and patching.** Upgrading or patching the in-kernel OVS dataplane required reloading the kernel module and rebooting the entire system, but an AF_XDP deployment only needs to restart OVS.

✔ **Easier development.** For SDN research or playing with OVS, the need to modify, compile, and load the kernel module makes development much harder, especially since it often means crashing

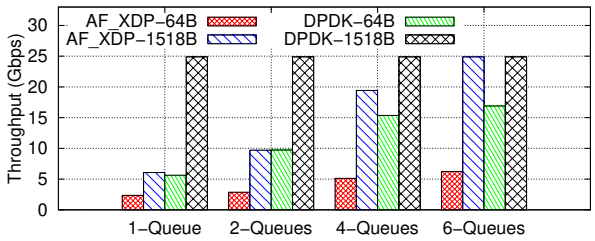

Figure 12: Throughput using multiple queues and cores of Physical-to-Physical (P2P) test using 25 Gbps network card.

and rebooting due to a bug. Testing and deploying in production is difficult too. Thanks to the lower difficulty level, we have seen increased interest in understanding new potential features that can be added at the datapath layer. This is the case for SIMD optimizations [75] using Intel AVX512 instructions to accelerate packet processing, and OVS-CD [27] that implements a better flow classification algorithm.

✔ **Easier troubleshooting.** Most of the traffic deployed in cloud data centers is tunneled and stateful. NSX usually programs tens of thousands of OpenFlow rules on each host running OVS bridges, with Geneve tunnel and connection tracking enabled. This complicated setup makes the verification of network configuration a very hard task. The userspace datapath makes troubleshooting easier. For example, a past bug in Geneve protocol parser [38], with an in-kernel datapath, might have triggered a null-pointer dereference that would crash the entire system. Now, such a bug will trigger only the health monitoring daemon to auto-restart OVS and create a core dump for root cause analysis.

✔ **Easier packaging and validation.** Every Linux kernel includes AF_XDP. OVS with AF_XDP does not need tight coordination with Linux kernel versions or DPDK versions. OVS today also integrates with several static and dynamic code analysis tools, such as Valgrind, Coverity, Clang memory/address sanitizers. The user/kernel split design requires running these analyses separately for the userspace and the kernel module. Furthermore, today, only a small portion of code is analyzed because most of the public Continuous Integration, Continuous Delivery (CI/CD) systems, e.g., Travis CI and Github Actions, don't support out-of-tree kernel module due to security concerns. This is because they do not provide root privileges that are needed to load kernel modules. The new OVS, being mainly a userspace process, fits better in the CI/CD pipeline resulting in a more robust release.

✔ **A better cross-platform design.** Customers of NSX sometimes run less-popular Linux distributions, such as IBM LinuxONE, Oracle Linux, Citrix XenServer, or deploy a kernel with customized security enhancement patches [51]. Any of these might change the kernel's internal behavior and break the OVS kernel module at compilation or loading time [80]. Our current userspace OVS datapath minimally depends on the operating system kernel, which makes porting to multiple platforms easier.

✗ **Some features must be reimplemented.** In-kernel OVS takes advantage of features of the Linux network stack such as its connection tracking firewall, NAT, and tunnel encapsulations. OVS

had to reimplement these in userspace. Traffic shaping and policing is still missing, so we currently use the OpenFlow meter action to support rate limiting, which is not fully equivalent. We are working on implementing better QoS features.

✗ **AF_XDP is not yet available for Windows.** OVS supports Windows through its own Windows-specific kernel driver and in userspace via Winsock. Given that Microsoft recently announces its eBPF for Windows project [45], using the uBPF [34] as JIT compiler and PREVAIL verifier [2], we hope that our work encourages Microsoft to add an AF_XDP-like means to pass packets between userspace and the kernel.

✗ **AF_XDP is not always the fastest option—yet.** OVS with AF_XDP and vhostuser interfaces is the fastest choice for applications inside VMs, and with XDP redirection it is also the fastest option for container-to-container UDP workloads. However, in-kernel OVS still outperforms OVS with AF_XDP for container-to-container TCP workloads. We expect this to change when TSO support becomes available for AF_XDP.

7 RELATED WORK

Several approaches have been proposed to solve the maintenance and extensibility burdens of the OVS Linux kernel datapath. In Section 2.2.2, we discussed the use of eBPF to allow for extensibility without requiring upstream Linux support. The possibility to enable new features using eBPF has also been demonstrated [54], which is compatible with a variety of OSes, e.g., VMware ESX, Linux KVM. However, the performance penalties caused by eBPF's in-kernel virtual machine outweigh its benefits [66, 72].

Performance and extensibility could be obtained by coupling DPDK with high-level programming languages such as P4. This is the case of PISCES [59], which extends OVS with the support of P4 language. Unfortunately, here the DPDK backend creates the compatibility issues discussed in Section 2.2.1, making this approach not appealing for real deployments. The same issue can be found in other proposals that fully bypass the kernel, such as Vector Packet Processing (VPP) [4], which recently has started to explore the possibility to support AF_XDP. This is also the case of Oko [10] that proposes to extend the OpenFlow protocol and allows for stateful filtering functionalities to be executed in userspace through an eBPF virtual machine.

Virtual Filtering Platform (VFP) [24] is the virtual switch for Azure SDN platform. VFP is tightly integrated with the Windows kernel to closely interact with the host network stacks. We assume VFP has similar issues as using OVS's kernel datapath in Linux environment. Finally, Snap [42] mentioned similar productivity and performance issue of in-kernel networking stack and proposes microkernel-like userspace networking system. Unlike Snap which bypasses the kernel and develops its own transport layer, our work focus on virtual switch and better integration and compatibility with Linux kernel community.

8 CONCLUSION

This paper shared our experience in supporting and running OVS, a state-of-the-art software adopted in cloud data center environments. Having the OVS datapath tightly integrated with the Linux kernel brought several drawbacks: (1) Features in OVS are limited by what Linux developers will accept in principle and then in implementation; (2) OVS upgrades or bug fixes that affect the kernel module can require updating the kernel and rebooting production systems; (3) Conventional in-kernel packet processing is now much slower than newer options.

We presented the evolution of the OVS dataplane alongside the techniques we employed to overcome the drawbacks of the previous implementations. The new OVS leverages recent advances in Linux networking, such as AF_XDP to quickly forward packets to userspace as soon as they reach the NIC driver.

Beside demonstrating comparable performance to OVS with DPDK, the solution proposed in this paper also dramatically reduces the efforts needed for validating new versions while making troubleshooting easier. The new code is already merged into the mainstream OVS repository.

ACKNOWLEDGMENTS

We would like to thank our shepherd Yashar Ganjali and the anonimous reviewers. Also, the many people contributing to the Open vSwitch community, providing directions to this work, and reviewing and evaluating its performance, including: Sujata Banerjee, Jesper Dangaard Brouer, Mihai Budiu, Eelco Chaudron, Magnus Karlsson, Niaz Khan, Toshiaki Makita, Ilya Maximets, David S. Miller, Yifeng Sun, and Björn Töpel. This work is partially supported by the UK's EPSRC under the projects NEAT (EP/T007206/1).

REFERENCES

[1] 6Wind. 2020. 6Wind Datacenter Networking. https://www.6wind.com/products/solutions/data-center-networking/.
[2] PREVAIL: a Polynomial-Runtime EBPF Verifier using an Abstract Interpretation Layer. 2021. uBPF. https://github.com/vbpf/ebpf-verifier.
[3] Ariel Adam and Amnon Ilan. 2019. How vhost-user came into being: Virtio-networking and DPDK. https://www.redhat.com/en/blog/how-vhost-user-came-being-virtio-networking-and-dpdk.
[4] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. 2018. High-Speed Software Data Plane via Vectorized Packet Processing. In *Communications Magazine, Volume: 56, Issue: 12*. IEEE.
[5] Lorenzo Bianconi. 2020. Introduce support for XDP programs in CPUMAP. https://lwn.net/Articles/826114/.
[6] Thomas Bittman, Philip Dawson, and Michael Warrilow. 2016. Gartner Magic Quadrant for x86 Server Virtualization Infrastructure. In *Gartner Research*.
[7] Brenden Blanco, Yonghong Song, et al. 2016. BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc/.
[8] Jesper Dangaard Brouer. 2016. XDP âĂŞ eXpress Data Path, Intro and future use-cases. http://people.netfilter.org/hawk/presentations/xdp2016/xdp_intro_and_use_cases_sep2016.pdf.
[9] Mihai Budiu. 2015. Compiling P4 to eBPF. https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4.
[10] Paul Chaignon, Kahina Lazri, Jérôme François, Thibault Delmas, and Olivier Festor. 2018. Oko: Extending Open vSwitch with stateful filters. In *Symposium on SDN Research (SOSR)*. ACM.
[11] Cisco Inc. [n.d.]. TRex: Realistic Traffic Generator. https://trex-tgn.cisco.com/.
[12] Jonathan Corbet. 2015. Improving Linux networking performance. https://lwn.net/Articles/629155/.
[13] Jonathan Corbet. 2018. Accelerating networking with AF_XDP. https://lwn.net/Articles/750845/.
[14] Jonathan Corbet. 2020. A medley of performance-related BPF patches. https://lwn.net/Articles/808503/.
[15] Linux Networking Documentation. 2020. AF_XDP: XDP_SKB and XDP_DRV Modes. https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html.
[16] DPDK. 2020. AF_XDP Poll Mode Driver. https://doc.dpdk.org/guides-20.05/nics/af_xdp.html.
[17] DPDK Community. 2018. DPDK - DataPlane Development Kit. http://www.dpdk.org/.
[18] DPDK Guide. [n.d.]. DPDK Tools User Guides. https://doc.dpdk.org/guides/testpmd_app_ug/.

[19] DPDK Guide. [n.d.]. Testpmd Application User Guide. http://fast.dpdk.org/doc/pdf-guides-18.08/tools-18.08.pdf.

[20] DPDK Guide. 2018. Virtio for Container Networking. https://doc.dpdk.org/guides/howto/virtio_user_for_container_networking.html.

[21] Ericsson. 2020. Cloud SDN. https://www.ericsson.com/en/portfolio/digital-services/cloud-infrastructure/cloud-sdn.

[22] Julia Evans. 2019. Writing eBPF tracing tools in Rust. https://jvns.ca/blog/2018/02/05/rust-bcc/.

[23] F-Stack. 2019. User Space Network Development Kit. http://www.f-stack.org/.

[24] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[25] Matt Fleming. 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[26] Linux Foundation. 2020. Open vSwitch upstream repository. https://github.com/openvswitch/ovs.

[27] Sameh Gobriel and Charlie Tai. 2017. OVS-CD: Optimizing Flow Classification for OVS using the DPDK Membership Library. https://www.openvswitch.org/support/ovscon2017/.

[28] DPDK Guide. 2019. Mbuf Library. https://static.sched.com/hosted_files/envoyconna18/e0/envoycon_dpdkvn.pdf.

[29] Stefan Hajnoczi. 2011. QEMU Internals: vhost architecture. http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html.

[30] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A software NIC to augment hardware. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155* (2015).

[31] Mark Haranas. 2019. Pat Gelsinger: Cisco ACI 'Bicycle' Will Never Match VMware NSX 'Lamborghini'. https://www.crn.com/news/networking/pat-gelsinger-cisco-aci-bicycle-will-never-match-vmware-nsx-lamborghini-.

[32] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.

[33] Magnus Karlsson and Björn Töpel. 2018. The path to DPDK speeds for AF XDP. In *Linux Plumbers Conference*.

[34] Rich Lane, Paul Chaignon, et al. 2021. uBPF. https://github.com/iovisor/ubpf.

[35] LF Projects. 2018. Tungsten Fabric Architecture. https://tungstenfabric.github.io/website/Tungsten-Fabric-Architecture.html.

[36] Linux bridge 2021. Linux bridge. https://wiki.linuxfoundation.org/networking/bridge.

[37] Hangbin Liu. 2020. xdp: add dev map multicast support. https://lwn.net/Articles/817582/.

[38] Alan Maguire. 2019. OVS iptunnel bug. https://github.com/openvswitch/ovs/commit/902c5ffd3360b05ad344f7f4f5ee0301ae331a5.

[39] Saeed Mahameed. 2020. XDP meta-data Acceleration. https://netdevconf.info/0x14/session.html?workshop-XDP.

[40] Saeed Mahameed and et al. 2020. Linux NetDev 0x14: XDP Workshop. https://netdevconf.info/0x14/session.html?workshop-XDP.

[41] Toshiaki Makita and William Tu. 2020. Linux NetDev 0x14: Fast OVS Datapath with XDP. https://netdevconf.info/0x14/session.html?talk-fast-OVS-data-path-with-XDP.

[42] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. 2019. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 399–413.

[43] John McNamara. 2017. API ABI Stability and LTS: Current state and Future. In *DPDK Summit Userspace*.

[44] Microsoft. 2020. Hyper-V Technology Overview. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview.

[45] Microsoft. 2021. eBPF for Windows. https://github.com/Microsoft/ebpf-for-windows.

[46] David Miller. 2015. net: Add STT support. https://marc.info/?l=linux-netdev&m=142275484322111&w=2.

[47] David S. Miller. 2012. Removing the Linux Routing Cache. http://vger.kernel.org/~davem/columbia2012.pdf.

[48] David S. Miller. 2016. Fast Programmable Networks & Encapsulated Protocols. https://netdevconf.info/1.2/session.html?david-miller-keynote.

[49] Paul Moore. 2017. Generic Netlink. https://lwn.net/Articles/208755/.

[50] Digital Ocean. 2020. OVS in the Cloud. http://www.openvswitch.org/support/ovscon2019/.

[51] Open Source Security, Inc. 2020. grsecurity. https://grsecurity.net/.

[52] OVS. 2020. Releases: Q: Are all features available with all datapaths? https://docs.openvswitch.org/en/latest/faq/releases/.

[53] OVS Community. 2018. Open vSwitch with DPDK. http://docs.openvswitch.org/en/latest/intro/install/dpdk/.

[54] Justin Pettit, Ben Pfaff, Joe Stringer, Cheng-Chun Tu, Brenden Blanco, and Alex Tessmer. 2018. Bringing Platform Harmony to VMware NSX. In *SIGOPS Operating Systems Review, Volume: 52, Issue: 1*. ACM.

[55] Ben Pfaff. 2011. Open vSwitch datapath developer documentation. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/openvswitch.rst.

[56] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open VSwitch. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[57] Greg Rose. 2018. compat: Add ipv6 GRE and IPV6 Tunneling. https://github.com/openvswitch/ovs/commit/c387d8177f20.

[58] Gregory Rose. 2020. Question about supporting the OVS out-of-tree kernel drivers. https://mail.openvswitch.org/pipermail/ovs-dev/2020-December/378831.html.

[59] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. PISCES: A Programmable, Protocol-Independent Software Switch. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[60] Pravin Shelar. 2015. Open vSwitch STT support. https://mail.openvswitch.org/pipermail/ovs-dev/2015-February/294420.html.

[61] Pravin B Shelar. 2014. openvswitch: Introduce flow mask cache. https://patchwork.ozlabs.org/project/netdev/patch/1406851074-1680-1-git-send-email-pshelar@nicira.com/.

[62] Jianfeng Tan, Cunming Liang, Huawei Xie, Qian Xu, Jiayu Hu, Heqing Zhu, and Yuanhan Liu. 2017. VIRTIO-USER: A new versatile channel for kernel-bypass networks. In *Proceedings of the Workshop on Kernel-Bypass Networks*. 13–18.

[63] Björn Töpel. 2020. Introduce AF_XDP buffer allocation API. https://lwn.net/Articles/821115/.

[64] Björn Töpel. 2020. Introduce preferred busy-polling. https://lwn.net/Articles/837010/.

[65] Linus Torvalds. 2005. GCC versus kernel stability. https://yarchive.net/comp/linux/gcc_vs_kernel_stability.html.

[66] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. 2017. Building an extensible Open vSwitch datapath. In *SIGOPS Operating Systems Review, Volume: 51, Issue: 1*. ACM.

[67] William Tu. 2018. AF_XDP support for veth. https://patchwork.ozlabs.org/cover/1015775/.

[68] William Tu. 2018. openvswitch: add erspan version I and II support. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=fc1372f89ffe.

[69] William Tu and Greg Rose. 2018. ERSPAN Support for Linux, Linux Plumber Conference. http://vger.kernel.org/lpc_net2018_talks/erspan-linux.pdf.

[70] William Tu, Fabian Ruffy, and Mihai Budiu. 2018. Linux Network Programming with P4. In *Linux Plumber Conference*.

[71] William Tu and Alexei Starovoitov. 2018. [RFC PATCH 00/11] OVS eBPF datapath. https://lists.iovisor.org/g/iovisor-dev/topic/22656941.

[72] William Tu, Joe Stringer, Yifeng Sun Sun, and Yi-Hung Wei. 2018. Bringing the Power of eBPF to Open vSwitch. In *Linux Plumber Conference*.

[73] William Tu, Yi-Hung Wei, and Ilya Maximets. 2020. Open vSwitch with AF_XDP. https://docs.openvswitch.org/en/latest/intro/install/afxdp/.

[74] Asanga Udugama. 2006. Manipulating the networking environment using RT-NETLINK. *Linux Journal* 145 (2006).

[75] Harry van Haaren. 2018. Applying SIMD Optimizations to the OVS Datapath. https://www.openvswitch.org/support/ovscon2018/5/1355-van-haaren.pdf.

[76] VMware. 2020. Update to VMwareâĂŹs per-CPU Pricing Model. https://www.vmware.com/company/news/updates/cpu-pricing-model-update-feb-2020.html.

[77] VMware. 2020. VMware NSX Data Center. https://www.vmware.com/products/nsx.html.

[78] Peter P. Waskiewicz Jr. and Neerav Parikh. 2018. Accelerating XDP Programs Using HW-based Hints. http://vger.kernel.org/lpc_net2018_talks/xdp-plumbers-2018.pdf.

[79] Yi-Hung Wei. 2018. datapath: compat: Backports nf_conncount. https://github.com/openvswitch/ovs/commit/744964326f6c.

[80] Yi-Hung Wei. 2018. datapath: compat: Fix compilation issue with grsecurity patch. https://github.com/openvswitch/ovs/commit/1556fcca6766.

[81] Yi Yang. 2019. OVS DPDK issues in Openstack and Kubernetes and Solutions. http://www.openvswitch.org/support/ovscon2019/.