Faster OVS Datapath with XDP

Toshiaki Makita
toshiaki.makita1@gmail.com
NTT

William Tu
u9012063@gmail.com
VMware NSBU

Abstract

XDP is fast and flexible, but difficult to use as it requires people to develop eBPF programs which often needs deep knowledge of eBPF. In order to mitigate this situation and to let people easily use an XDP-based fast virtual switch, we introduce Open vSwitch (OVS) acceleration by XDP.

This has been already partly realized by recently introduced OVS netdev type, afxdp. Afxdp netdev type bypasses kernel network stack through AF_XDP sockets and handles packets in userspace. Although its goal, bringing XDP's flexibility to OVS datapath by enabling us to update the datapath without updating kernel, was different from acceleration, afxdp is basically faster than OVS kernel module. However, compared to native XDP in kernel (i.e. non-AF_XDP), it is more difficult to get optimal performance since softirq and userspace process need to work in concert, and it also imposes some overhead like skb conversion on packet transmission due to missing zero-copy support in drivers.

We are trying to bring even faster data path to OVS by attaching native in-kernel XDP programs which implement subset of OVS datapath. We propose two different approaches to attach such XDP programs from OVS userspace daemon. One is to attach XDP from kernel through UMH (user mode helper), reusing TC hardware acceleration interface. The other is to attach XDP directly from OVS userspace daemon. The former has an advantage in that it enables XDP for other flow-based network functionalities than OVS at the same time. The latter has more flexibility in that users can attach arbitrary XDP programs which may have minimal computational cost for their use cases. In this article we discuss advantages and drawbacks of those two approaches, and show how much the proposed approach improve OVS datapath performance.

1. Introduction

1.1 XDP use-case: Performance improvement of kernel network features

XDP, eXpress Data Path [1, 2, 6], is a super-fast and flexible kernel networking path. One use-case of it is to replace existing network functions with XDP-based implementation. Open vSwitch (OVS) is a typical example of such network functions.

However, it is often difficult for XDP users to develop large eBPF programs like OVS datapath. Especially there are a lot of restrictions in eBPF verifier so developers tends to encounter errors on loading programs into kernel, which is time-consuming as the workaround is specific to eBPF and requires deep knowledge of eBPF. This can discourage people from using XDP despite its performance and flexibility.

One possible solution of this is to provide an XDP program which is confirmed to work correctly as a replacement of OVS existing datapath modules.

1.2 Open vSwitch and XDP

OVS is widely used in virtualized data center environments as a software switching layer for a variety of operating systems. As OVS is on the critical path, concerns about performance affect the architecture and features of OVS. For complicated configurations, OVS's flow-based configuration can require huge rule sets. The key to OVS's forwarding performance is flow caching. As shown in Figure 1(a), OVS consists of two major components: a slow path userspace process and a fast path caching component, called the *datapath*.

When a packet is received by the NIC, the host OS does some initial processing before giving it to the OVS datapath. The datapath first parses the packet to extract relevant protocol headers and stores it locally in a manner that is efficient for performing lookups, then it uses this information to look into a flow cache to determine the actions to apply to the packet. If there is no match in the flow table, the datapath passes the packet from the kernel up to the slow path, `ovs-vswitchd`, which maintains the full determination of what needs to be executed to modify and forward the packet correctly. This process is called packet *upcall* and usually happens at the first packet of a flow seen by the OVS datapath. If the packet matches in the datapath's flow cache, then the OVS datapath executes the corresponding actions from the flow table lookup result and updates its flow statistics.

Currently OVS has three datapath implementations: Linux kernel datapath, userspace netdev datapath, and Windows kernel datapath. OVS-DPDK [12] is a community project that uses the userspace netdev datapath with DPDK library as a kernel bypassing packet I/O. With many years of optimization efforts from the community, it shows the best performance among the three datapath implementations. However, the system requirements of deploying DPDK and configurations make DPDK not suitable for every use case. Linux AF_XDP is a Linux kernel's socket interface that provides high packet I/O rate. Recent version of OVS adds support for AF_XDP coupled with the userspace netdev datapath and shows performance closed to OVS-DPDK. This is because it is using the same datapath implementation as OVS-DPDK uses, with the only difference that its packet I/O driver is AF_XDP socket, not DPDK's device driver.

The current implementation of AF_XDP on OVS, as shown in Figure 1(b), loads a minimum XDP program in the network device and forwards all packets from the XDP driver hook point to the userspace datapath, without taking any advantage of the fast and flexible XDP packet processing in kernel. In the article, we propose a faster XDP datapath to OVS by attaching native in-kernel XDP programs which implement subset of OVS datapath, shown in Figure 1(c). We propose two different approaches to attach such XDP programs from OVS userspace daemon. One is to attach XDP from kernel through UMH (user mode helper), reusing TC HW acceleration interface. The other is to attach XDP directly from OVS userspace daemon. The former has an advantage in that it en-

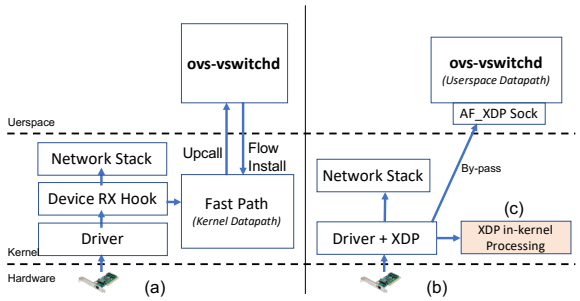


Figure 1: *The forwarding plane of OVS consists of two components: ovs-vswitchd handles the complexity of the OpenFlow protocol, while the datapath, or fast path, acts as a caching layer to optimize the performance. (a) shows the datapath implemented as a kernel module, processing packets in the kernel. (b) shows the AF_XDP by-passed approach, using an XDP program to redirect packets to a userspace datapath running in ovs-vswitchd. (c) shows our proposed approach, loading the XDP program into kernel and processing the flows by implementing parse, lookup, and actions in XDP without always forwarding packets to userspace datapath.*

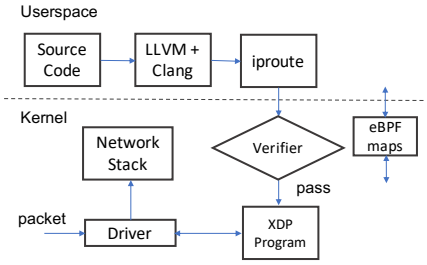


Figure 2: *The workflow of XDP eBPF development process and its packet flow. The eBPF program is compiled by LLVM+clang and is loaded into the kernel using iproute. The kernel runs the program through a verification stage, and subsequently attaches the program to the XDP ingress hook point. Once successfully loaded, an incoming packet received by the driver will execute the eBPF program.*

ables XDP for other flow-based network functionalities than OVS at the same time. The latter with OVS userspace daemon shows more flexibility in that users can attach arbitrary XDP programs which may have minimal computational cost for their use cases. We present design, implementation, and evaluation of the system in the following sections.

2. Background

2.1 eBPF and XDP

eBPF, extended Berkeley Packet Filter, enables userspace applications to customize and extend the Linux kernel's functionality. Programs written for eBPF are compiled to eBPF bytecode, which is an instruction set that runs on the eBPF virtual machine inside the kernel. On many platforms, the bytecode is run through a JIT to generate code that runs on the CPU architecture of the system, which allows near native performance. eBPF programs are executed by attaching them to hook points that have been added throughout the kernel. A popular use-case for eBPF is tracing, which allows administrators to write small eBPF programs and attach them to subsystems such as the file system and CPU scheduler to get fine-grained performance information on a live system. However, eBPF is not limited to tracing; larger programs can also be written and applied to different networking subsystems.

XDP [1, 2, 6] is the name for an eBPF program hook point that executes within the network driver, the lowest level of the network stack. XDP demonstrates high performance close to the line rate of the device, since programs attached to the XDP hook point are triggered immediately in the network device driver's packet receiving code path. For this reason, an eBPF program in XDP can only access the packet data and limited metadata, since the kernel has not processed the packet as much as hook points later in the network stack, such as *tc*, the traffic control subsystem.

Figure 2 shows the typical workflow for installing an eBPF program to the XDP hook points, and how packets trigger eBPF execution. Clang and LLVM take a program written in C and compile it to the eBPF instruction set, and then emit an ELF file that contains the eBPF instructions. An eBPF loader, such as iproute, takes the ELF file, parses its programs and map information, and issues BPF syscalls to load the program. If the program passes the BPF verifier, then it is attached to the hook point (in this case, XDP), and subsequent packets through the XDP ingress hook will trigger execution of the eBPF programs.

2.2 AF_XDP

AF_XDP is a new Linux address family that aims for high packet I/O performance. It enables another way for a userspace program to receive packets from kernel through the socket API. For example, currently, creating a socket with address family AF_PACKET, userspace programs can receive and send raw packets at the device driver layer. Although the AF_PACKET family has been used in many places, such as tcpdump, its performance does not keep up with recent high speed network devices, such as 40Gbps and 100Gbps NICs. A performance evaluation [7] of AF_PACKET shows fewer than 2 million packets per second using a single core.

AF_XDP was added to the Linux kernel in version 4.18 [3]. The core idea behind AF_XDP is to leverage the XDP eBPF program's early access to the raw packet and provide a high speed channel from the NIC driver directly to a userspace socket interface. In other words, the AF_XDP socket family connects the XDP packet receiving/sending path to the userspace, bypassing the rest of the Linux networking stack. An AF_XDP socket, called XSK, is created by using the normal socket() system call, which makes integrating into Linux-based system easier.

2.3 OVS Userspace DP with AF_XDP

OVS's architecture defines multiple interface types to allow alternative implementations of various parts of the system. The datapath interface (dpif) allows different datapaths to be implemented. The traditional one provides an interface used by kernel datapath implementations. However, OVS provides a userspace datapath interface implementation called *dpif-netdev*. dpif-netdev is designed to be agnostic to how the network device accesses the packets, by an abstraction layer called netdev. We implement a new netdev type for AF_XDP, which receives and transmits packets using XSK. We insert an XDP program and an eBPF map that interacts with XDP program to forward packets to the AF_XDP socket. Once the AF_XDP netdev receives a packet, it passes the packet to the dpif-netdev for packet processing, as shown in Figure 1(b).

The current OVS implementation does not process packet in the context of XDP, but it simply forwards every packet to the OVS Userspace DP, the dpif-netdev. This paper proposes offloading the packet processing into the XDP context, using the existing offload API used by TC-flower and DPDK rte_flow. The result shows better integration with kernel and better performance.

2.4 OVS Hardware offload mechanism

As described above, OVS has flow offload mechanism using TC-flower. TC-flower is an in-kenrel flow classifier feature which can offload flows to hardware NIC, so OVS TC-flower offload is used for flow offload to hardware.

Figure 3 shows how TC-flower is used to offload flows. The offload for each flow is triggered when the flow is installed in

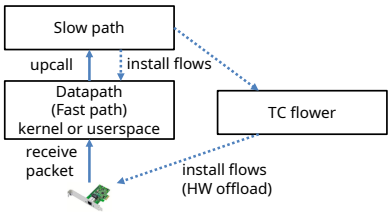


Figure 3: *OVS Hardware offload using TC flower*

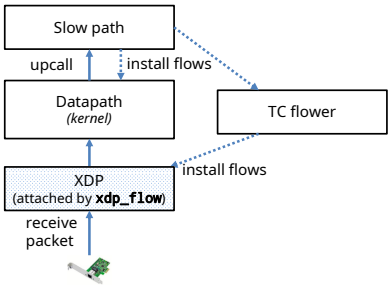


Figure 4: *xdp_flow with OVS*

datapath flow table if hardware offload feature of OVS is enabled. This typically happens when datapath cannot find an entry for a received packet and does `upcall`.

The TC offload functionality installs the offloaded flow in kernel's TC flower classifier. TC flower, if configured to offload to hardware, installs the flow in NIC hardware. After that, any packets that match the flow installed in the NIC is handled and forwarded in the NIC instead of CPU (datapath).

We use this offload mechanism in both of two proposed approaches.

3. Faster Open vSwitch Datapath with XDP

As desribed in section 1, by providing an XDP program which is confirmed to work correctly as a replacement of OVS existing datapath, we can provide an easy way to make OVS datapath faster. AF_XDP netdev partially implements this idea by using existing userspace datapath through AF_XDP. However, with this model userspace daemon and XDP in kernel needs to work in concert with each other, which can result in suboptimal performance.

We propose two approaches for introducing XDP programs running fully in kernel context to improve performance with XDP.

3.1 xdp_flow

`xdp_flow` [10] is an in-kernel infrastructure to offload flows to XDP. xdp_flow is a generic flow offload engine using XDP. Any flow-based in-kernel features such as TC flower and nftables can be offloaded to XDP by xdp_flow through flow hardware offload mechanism in kernel. OVS is also offloadable through TC flower because OVS uses TC flower as an offload engine.

Figure 4 shows how xdp_flow works for OVS. xdp_flow requires OVS to enable TC offload. When xdp_flow is enabled, flows offloaded from OVS to TC flower are offloaded to XDP from TC instead of NIC hardware through TC hardware offload mechanism.

Figure 5 shows more details of how xdp_flow uses in-kernel hardware flow offload mechanism. xdp_flow is implemented as a driver for hardware flow offload. When it receives a request for offload, it installs the flow in XDP instead of hardware NIC. This model requires a feature to attach XDP programs to NICs from kernel. We implemented such a feature by using user-mode blobs [5, 14] introduced for bpfilter [4, 13]. User-mode blobs are user mode programs embedded in kernel. They can be run as processes through user-mode helper (UMH [8]) mechanism. In short, kernel modules can run embedded user-mode programs through

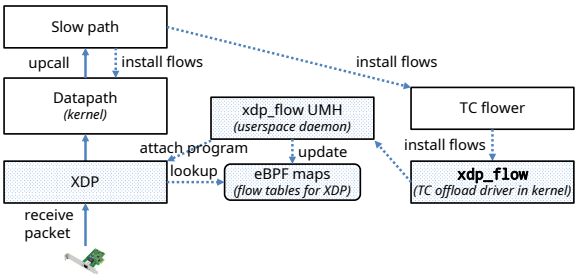


Figure 5: *xdp_flow Details*

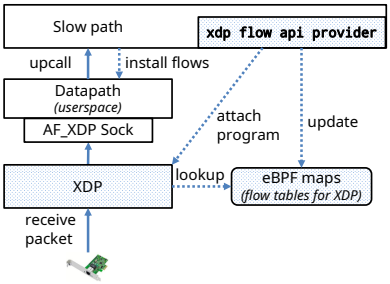


Figure 6: *xdp flow api provider*

user-mode blobs mechanism. With this mechanism, xdp_flow launch a daemon program on loading its kernel module. The daemon communicates with xdp_flow kernel module, and is responsible for handling any eBPF manipulation including attaching XDP programs and updating eBPF maps used for flow tables in the XDP programs. The attached XDP program has maps for flow tables and handle packets based on the table. If any table miss happens, e.g. the flow for the packet is not supported by the XDP program, the packet is passed to the upper layer using XDP_PASS action, and then the packet will be handled by OVS kernel datapath as normal.

With this xdp_flow mechanism, users can easily make use of XDP because the difficult part, making XDP programs and maintenance of flow tables in XDP, is automatically handled by kernel. OVS users just need to enable TC offload in OVS and xdp_flow in kernel. xdp_flow can be enabled by `ethtool` command per network interfaces, so it is simple. Once it is enabled, OVS can be used as normal, and XDP offload for each flow is done automatically.

The advantage of this model is that it enables flow offload not only for OVS but also for other flow-based features like TC flower (without OVS) and nftables.

However, it has some drawbacks as well mainly because the embedded program is not modifiable. Users cannot customize the program for thier use-cases. XDP is fast when it has minimal functionalities for its use-case, so non-modifiable program is unfavorable performance-wise. The embedded program also cannot work with AF_XDP netdev in OVS as AF_XDP programs need to be loaded from processes which use the AF_XDP socket connected to xsk maps in the XDP programs.

These drawbacks can be overcome by handling XDP programs from OVS daemon ovs_vswitchd, sacrificing the merit of code sharing with other flow-based kernel features. Xdp flow api provider illustrated in the next section is based on this idea.

3.2 xdp flow api provider

Flow api provider means offload driver of OVS. Our second approach is `xdp flow api provider` [11], which is an OVS offload driver using XDP.

The idea is similar to xdp_flow but it directly uses OVS flow offload functionality instead of using TC flower offload driver (Figure 6). Another difference is that it does not use embedded or unmodifiable XDP programs. Instead of embedded programs, we pro-

vide a reference XDP program. Users can customize the XDP program and load it instead of the original one. Xdp flow api provider in OVS daemon detects whether the loaded program has necessary features like maps with certain names for flow tables, and if it determines the program is usable for flow offload, the driver starts offloading flows to the program.

This mechanism allows the program to work with AF_XDP netdev as well as allowing for the use of a minimal program for each use-cases of individual users.

3.3 Reference XDP program for xdp flow api provider

As described in the previous section, we provide a reference XDP program in OVS tree for xdp flow api provider. It is meant for general OVS use, but users can remove any unnecessary code like key parsers or actions, or remove unneeded keys in flow hash table to make it faster.

Even without keys removal in code, it can reduce flow table hash key size by just redefining the size by a macro with a smaller value. The program uses miniflow mechanism which is originally used in ovs_vswitchd. Miniflow can dynamically compress the key size by removing unused keys. Although it is impractical to dynamically change hash key size online due to eBPF map restriction, since it is rare to use the combination of all the keys, the maximum needed key size will be far less than full key size. Thus users can just choose a reasonable key size for flow hash tables and reduce the overhead.

3.4 Performance

In this section we show packet forwarding throughput of xdp flow api provider compared to AF_XDP netdev and traditional kernel datapath. We used 2 machines connected through an ethernet switch. Each machine has:

- CPU: Intel Xeon Silver 4114, 2.20 GHz.
- NIC: Intel XXV710 25-Gigabit, i40e driver.
- kernel: 5.5.5
- Open vSwitch: 2.13 + our XDP offload patch

spectre_v2 mitigation is disabled on each machine.

One machine is used for the sender and the other is used for the receiver. The sender sends one flow UDP packets. The receiver receives them on an i40e NIC, and forwards them to another interface. Packets are discarded after forwarding.

The tests was performed in two configurations.

- i40e-veth

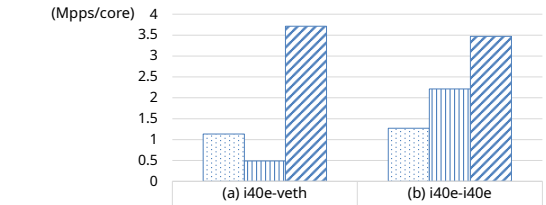
  The receiver forwards packets from i40e NIC to veth interface. The peer of the veth interface has an XDP program which only does XDP_DROP for any packets. Counts the number of dropped packets in the XDP program.

- i40e-i40e

  The receiver forwards packets from i40e NIC to i40e NIC. The forwarded packets are discarded on another machine. Counts the number of transmitted packets on the sending i40e NIC of the receiver machine by using `ip -s -s link show` command.

In each test we used pktgen to generate traffic from the sender machine. The sending rate is approximately 37 Mpps, almost line rate of 25 Gbit Ethernet. All packets belong to one flow of UDP (the same source/destination IP and port). The receiver always receives packets on a particular CPU core.

Figure 7 shows the results. `kernel module` is traditional kernel datapath. `afxdp` uses netdev (userspace) datapath and netdev of type `afxdp` with xdp-mode `native-with-zerocopy`.



	(a) i40e-veth	(b) i40e-i40e
kernel module	1.134	1.27
afxdp	0.49	2.214
xdp flow api provider	3.713	3.471

Figure 7: *Throughput of AF_XDP and XDP Flow Api Provider*

`xdp flow api provider` is what we proposed in the previous section. It uses netdev (userspace) datapath and netdev of type `afxdp-nonpmd` with xdp-mode `native` and xdp flow offload. `xdp flow api provider` uses different netdev type and xdp-mode from `afxdp` because the best configuration is different from AF_XDP without XDP offload. For XDP offload, zerocopy feature of AF_XDP has non-negligible overhead due to packet copy on XDP_REDIRECT, so use native XDP without zerocopy. Note that veth does not have zerocopy feature as of kernel 5.5.5, so it is always disabled on veth. `afxdp` netdev type runs poll mode driver (PMD) in userspace to boost AF_XDP performance. This does not improve XDP offload performance and consumes more CPU time, so used `afxdp-nonpmd` for XDP offload. The unit of the results is Mpps per core, which is 1,000,000 packets per second per core. Since netdev type `afxdp` uses two cores for one flow (one is for PMD and the other is for softirq), we divided its results by two. Note that we can use another configuration which uses only one core for `afxdp` without XDP offload, where both of PMD and softirq run in one core. However, this decreases the performance in pps/core even with using `use-need-wakeup` feature and/or `afxdp-nonpmd`, both of which can improve performance in such a situation.

The results show that in each case `xdp flow api provider` improves performance of AF_XDP. Especially with veth AF_XDP has less performance than kernel, but XDP offload shows nearly the same performance with non-veth case (i40e-i40e). We did not identify the root cause of the low performance of AF_XDP with veth, but one possible reason is that veth does not have zerocopy feature.

4. Future Work

We are now mainly working on upstreaming xdp flow api provider. Although we have a working patch set of basic functionality [11], there are a couple of future work.

- Support for more keys and actions

  Currently we have implemented very basic keys and actions with flow api provider, e.g. mac address, vlan id, ip address, l4 ports match and vlan push/pop, output action. In order to apply xdp flow api provider to wide range of use-cases, we should more keys and actions like tunneling-related ones.

- Further performance improvement

  XDP_REDIRECT from i40e to veth can acheive 10 Mpps/core if there is no extra logic in an XDP program [9]. While testing xdp_flow, we found that if we can remove some overhead in kernel the throughput can be up to 5.2 Mpps/core [10]. This should also apply to xdp flow api provider, so there is room to improve the performance at least at kernel side XDP infrastructure.

- Hardware offload

  Although XDP mechanism itself has hardware offload ability, currently the reference XDP program cannot be offloaded to

NIC hardware. This is because it uses map-in-map feature of eBPF to support subtables per flow mask for the datapath flow table, and map-in-map is not offloadable currently due to XDP offlaod restriction. If XDP offload can support map-in-map, or if we can support XDP programs without map-in-map, hardware offload through XDP may be possible in the future.

5. Related Work

5.1 OVS-DPDK

OVS has another fast datapath, OVS-DPDK [12]. While DPDK is in some cases faster [6] than XDP, it is not always the best choice due to some limitations. For example it does not support veth interfaces which are often used in containers environment. Instead it supports virtio-user ports for high speed container networking, but it requires containers to run DPDK programs. Another limitation is that it exclusively owns a network interface. XDP, including AF_XDP, is more cooperative with kernel, and can pass received packets to upper layer kernel network stack. Thus, network acceleration using XDP like xdp flow api provider can be more flexible at the same time the performance is comparable with DPDK.

5.2 bpfilter

Bpfilter [13] is an in-kernel feature that automatically load eBPF programs, attached to TC or XDP, which implements iptables rules configured in kernel. xdp_flow uses a similar mechanism with bpfilter in that both of them automatically install the same functionality with an existing kernel feature using eBPF through user-mode blobs [14]. One big difference is that bpfilter dynamically assembles an eBPF program in its UMH based on configured iptables rules while xdp_flow uses an embedded non-modifiable eBPF program for XDP. The reason xdp_flow does not dynamically assemble eBPF programs is that flow insertion is typically triggered by `upcall`. Upcall is caused by packet reception, so this rate is not controllable by the OS. Also its rate is often high, so it is not very practical to dynamically assemble eBPF programs on upcall. Instead, xdp_flow uses eBPF hash maps to implement flow tables. Compared to sequential access, which is O(n), used by eBPF programs dynamically assembled by bpfilter, the complexity of hash maps access is O(1). So in large flow table case, the approach used by xdp_flow should have less overhead. On the other hand non-modifiable eBPF program has disadvantage that minimal programs for each use-case is not possible, but it can be resolved by xdp flow api provider.

6. Conclusion

We proposed two approaches to offload flows to XDP. Currently we are working on the second approach, xdp flow api provider, and it shows solid performance improvement. With this feature people can use XDP to make OVS faster without learning extensive eBPF knowledge to write XDP programs on thier own.

References

[1] XDP: eXpress Data Path. `https://www.iovisor.org/technology/xdp`, 2018.

[2] Jesper Dangaard Brouer. XDP – eXpress Data Path, intro and future use-cases. *NetDev 1.2*, 2016.

[3] Jonathan Corbet. Accelerating networking with AF_XDP. `https://lwn.net/Articles/750845/`, 2018.

[4] Jonathan Corbet. Bpf comes to firewalls. `https://lwn.net/Articles/747551/`, Feb 2018.

[5] Jonathan Corbet. Bpfilter (and user-mode blobs) for 4.18. `https://lwn.net/Articles/755919/`, May 2018.

[6] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *CoNEXT'18: International Conference on emerging Networking EXperiments and Technologies*. ACM Digital Library, 2018.

[7] Magnus Karlsson, Björn Töpel, and John Fastabend. AF_PACKET V4 and PACKET_ZEROCOPY. In *Netdev Conference 2.2*, 2017.

[8] Linux Kernel. call_usermodehelper_exec. `https://www.kernel.org/doc/htmldocs/kernel-api/API-call-usermodehelper-exec.html`.

[9] Toshiaki Makita. veth: Driver XDP. `https://patchwork.ozlabs.org/project/netdev/cover/1533283098-2397-1-git-send-email-makita.toshiaki@lab.ntt.co.jp/`, Aug 2018.

[10] Toshiaki Makita. xdp_flow: Flow offload to XDP. `https://lwn.net/Articles/802653/`, Oct 2019.

[11] Toshiaki Makita. XDP offload using flow API provider. `https://mail.openvswitch.org/pipermail/ovs-dev/2020-June/372184.html`, Jun 2020.

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

[13] Alexei Starovoitov. net: add skeleton of bpfilter kernel module. `https://patchwork.ozlabs.org/project/netdev/patch/20180522022230.2492505-3-ast@kernel.org/`, May 2018.

[14] Alexei Starovoitov. umh: introduce fork_usermode_blob() helper. `https://patchwork.ozlabs.org/project/netdev/patch/20180522022230.2492505-2-ast@kernel.org/`, May 2018.