Bringing the Power of eBPF to Open vSwitch

William Tu u9012063@gmail.com
Joe Stringer stringerjoe@vmware.com
Yifeng Sun pkusunyifeng@gmail.com
Yi-Hung Wei yihung.wei@gmail.com
VMware Inc.

Abstract

eBPF is an emerging technology in Linux kernel with the goal of making Linux kernel extensible by providing an eBPF virtual machine with safety guarantee. Open vSwitch, OVS, is a software switch running majorly in Linux operating systems and interacts with Linux kernel module, the openvswitch.ko. In this paper we describe our efforts to make use of eBPF technology in OVS. We present the design, implementation, and evaluation of two projects: the OVS-eBPF project and the OVS-AFXDP project. OVS-eBPF has the goal of re-writing existing flow processing features in openvswitch kernel module into eBPF code, by attaching eBPF program to Linux TC. On the other hand, the OVS-AFXDP project aims to by-passing the kernel using AF_XDP socket and moves most of the flow processing features into userspace.

1. Introduction

eBPF, extended Berkeley Packet Filter, enables userspace applications to customize and extend the Linux kernel's functionality. It provides flexible platform abstractions for network functions, and is being ported to a variety of platforms. Among the various different ways of using eBPF, OVS has been exploring the power of eBPF in three: 1) in-kernel flow processing by attaching eBPF programs to TC, 2) offloading a subset of processing to XDP (eXpress Data Path), and 3) moving the flow processing to userspace by using AF_XDP.

Attaching eBPF to TC, OVS-eBPF project, started first with the most aggressive goal: we planned to re-implement the entire features of OVS kernel datapath under net/openvswitch/* into eBPF code. We worked around a couple of limitations, for example, the lack of TLV (Type-Length-Value) support led us to redefine a binary kernel-user API using a fixed-length array; and without a dedicated way to execute a packet, we created a dedicated device for user to kernel packet transmission, with a different BPF program attached to handle packet execute logic. Currently, OVS-eBPF can satisfy most of the basic feature requirement for flow processing, including tunneling protocol support. But more complicated processing such as connection tracking, NAT, (de)fragmentation, and ALG are still under discussion.

Projects	OVS-eBPF	OVS-AFXDP
Packet Rate	Low	High
Driver Support	No	Yes
Minimal Kernel Req.	4.18	Unknown
BPF Code Size	Large	Minimal
Extensibility	Low	High

Table 1: Comparison of the OVS-eBPF project and OVS-AFXDP project.

Moving one layer below TC is called XDP (eXpress Data Path), a much faster layer for packet processing, but with almost no extra packet metadata and limited BPF helpers support. Depending on the complexity of flows, OVS can offload a subset of its flow processing to XDP when feasible. However, the fact that XDP has fewer helper function support implies that either 1) only very limited number of flows are eligible for offload, or 2) more flow processing logic needed to be done in native eBPF code.

XDP provides another way for interacting with userspace programs, called AF_XDP. AF_XDP is another socket interface for control plane and a shared memory API for accessing packets from userspace application. OVS today has another full-fledged datapath implementation in userspace, called dpif-netdev, mostly used by DPDK community. OVS-AFXDP project treats the AF_XDP as a fast packet-I/O channel, and move most of the flow processing into the userspace OVS.

A rough comparison of the two project is shown in Table 4 This paper describes the design, implementation, and evaluation of the OVS-eBPF and OVS-AFXDP projects.

2. Background

OVS is widely used in virtualized data center environments as a software switching layer inside various operating systems, including FreeBSD, Windows Hyper-V, Solaris and Linux. As shown in Figure 1, the architecture of OVS consists of two major components: a slow path and a fast path. OVS begins processing packets in its datapath, the fast path, shortly after the packet is received by the NIC in the host OS. The OVS datapath first performs packet parsing to extract relevant protocol headers from the packet and store it locally in a manner that is efficient for performing lookups (flow key), then it uses this information to look into the match/action cache (flow table) and determines what needs to be done

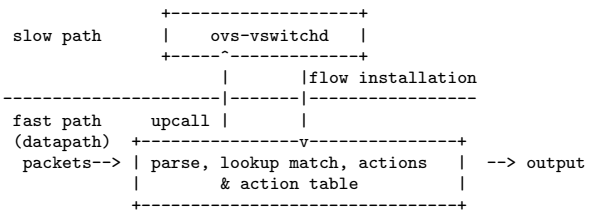

Figure 1: The forwarding plane of OVS consists of two components; ovs-vswitch handles the complexity of the OpenFlow protocol, and the datapath acts as a caching layer to optimize the performance. A flow missed by the match/action table in the datapath triggers an *upcall*, which forwards the information to ovs-vswitchd. ovs-vswitchd installs an appropriate flow entry into the datapath's match/action table.

for this 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 this flow table, then the OVS datapath executes its corresponding actions from the flow table lookup result and updates its flow statistics.

In this model, the `ovs-vswitchd` determines how the packet should be handled, and passes the information to the datapath inside the kernel using a Linux generic netlink interface. Over the years the OVS datapath features evolved. The initial OVS datapath used a microflow cache for its flow table, essentially caching exact-match entries for each transport layer connection's forwarding decision. And in later versions, two layers of caching were used: a microflow cache and a megaflow cache, which caches forwarding decisions for traffic aggregates beyond individual connections. In recent versions of OVS, datapath implementations include features such as connection tracking, stateful network address translation, and support for layer 3 tunneling protocols.

2.1 OVS Forwarding Model

2.2 eBPF Basics

Berkeley Packet Filter, BPF, is an instruction set architecture proposed by Steven McCanne and Van Jacobson in 1993 [1]. BPF was designed as a generic packet filtering solution and is widely used by every network operator today, through the well-known tcpdump/wireshark applications. A BPF interpreter is attached early in the packet receive call chain, and it executes a BPF program as a list of instructions. A BPF program typically parses a packet and decides whether to pass the packet to a userspace socket. With its simple architecture and early filtering decision logic, it can execute this logic efficiently.

For the past few years, the Linux kernel community has improved the traditional BPF (now renamed to classic BPF, cBPF) interpreter inside the kernel with additional instructions, known as extended BPF (eBPF). eBPF was introduced with the purpose of broadening the programmability of the Linux kernel. Within the kernel, eBPF instructions run in a virtual machine environment. The virtual machine provides a few registers, stack space, program counter, and a way to interact with the rest of the kernel through a mechanism called helper functions. Similar to cBPF, eBPF operates in an event-driven model on a particular hook point; each hook point has its own execution *context* and execution at the hook point only starts when a particular type of event fires. A BPF program is written against a specific context. For example, a BPF program attached to a raw socket interface has a context which includes the packet, and the program is only triggered to run when there is an incoming packet to the raw socket.

The eBPF virtual machine provides a completely isolated environment for its bytecode running inside; in other words, it cannot arbitrarily call other kernel functions or access into memory outside its own environment. To interact with the outside world, the eBPF architecture white-lists a set of helper functions that a BPF program can call, depending on the *context* of the BPF program. For example, a BPF program attached to raw socket in a packet context could invoke VLAN push or pop related helper functions, while a BPF program with a kernel tracing context could not.

To store and share state, eBPF provides a mechanism to interact with a variety of key/value stores, called *maps*. eBPF maps reside in the kernel, and can be shared and accessed from eBPF programs and userspace applications. eBPF programs can access maps through helper functions, while userspace applications can access maps through BPF system calls. There are a variety of map types for different use cases, such as hash tables or arrays. These are created by a userspace program and may be shared between multiple eBPF programs running in any hook point.

Finally, eBPF tail call [**?**] is a mechanism allowing one eBPF program to trigger execution of another eBPF program, providing users the flexibility of composing a chain of eBPF programs with each one focusing on particular features. Unlike a traditional function call, this tail call mechanism calls another program without returning back to the caller's program. The tail call reuses the caller's stack frame, which allows the eBPF implementation to minimize call overhead and simplifies verification of eBPF programs.

2.3 XDP: eXpress Data Path

There are several hook points where eBPF programs may be attached in recent Linux kernels. XDP is still an eBPF program, but its attachment point is at the lowest level of the network stack. Due to its unique attachment point, the XDP has its own *context*; the input parameters to the XDP eBPF program and return values have different meanings than the TC eBPF program. XDP shows performance closed to line rate of the device, because the XDP program is triggered immediately at the network device driver's packet receiving code path. Due to its lowest hook point at the networking stack, XDP input parameter has only pointer to the beginning and

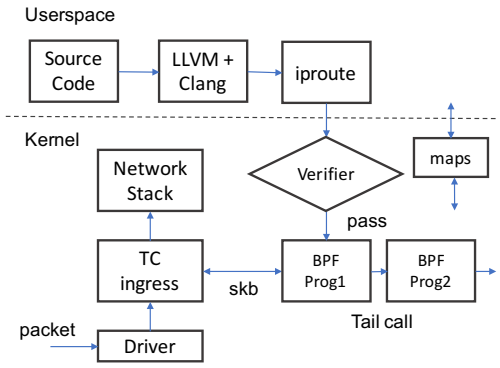

Figure 2: *The workflow of TC and XDP eBPF development process and its packet flow. The eBPF program compiled by LLVM+clang is loaded into the kernel using iproute. The kernel runs the program through a verification stage, and subsequently attaches the program to the TC/XDP ingress hook point. Once successfully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

end of the packet data plus a few metadatas. XDP also supports accessing to eBPF maps and tail calls, but with much less number of helper functions than the eBPF program at TC hook.

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

2.4 AF_XDP Socket

AF_XDP is a new Linux address family that aims for high packet I/O performance. Traditionally, a userspace program receives packets from kernel through the socket API. By creating a socket with address family such as AF_PACKET, userspace programs can receive/send the raw packets at the device driver layer. Although the AF_PACKET family has been using in many places such as tcpdump, its performance does not catch up with the recent high speed network devices, such as 40G/100G NICs. Performance evaluation [**?**] of AF_PACKET shows less than 2 million packets per second using single core.

AF_XDP was proposed and upstreamed to Linux kernel since 4.18 [**?**] The core idea behind the AF_XDP is to leverage the XDP eBPF program's early access to the raw packet, and create a high speed channel from the XDP directly to a userspace socket interface. In other word, AF_XDP socket family connects the XDP packet receving/sending path to the userspace, bypassing the rest of the Linux networking stacks. An AF_XDP socket, called XSK, is create using the normal socket() system call. Unlike AF_PACKET which uses the send() and receive() syscalls with packet buffer as parameter, XSK introduces two rings in userspace: the RX ring and the TX ring. The userspace program using XSK needs to properly configure and maintain the RX and TX ring structure in order to receive and send packets. In addition, all the packet data buffers used in TX/RX ring are allocated from a specific memory area called UMEM which consists of a number of fixed size chunks. The UMEM also has two rings: the FILL ring and the COMPLETION ring. A descriptor in RX/TX ring or in FILL/COMPLETION ring points to the element in UMEM by its address. The address is not system's virtual or physical address but simply an offset within the entire UMEM memory region.

For example, to receive packets, a set of descriptors pointing to empty packet buffer needs to be filled into the FILL ring. When packets arrives, the userspace program checks the RX ring, fetches the packet data from the desciptors, and refills the empty buffer to the FILL ring structure, in order to receive new incoming packets. For sending packets, a set of descriptors pointing the packet buffer with contents filled to the TX ring then issus send() system call. It is up to the userspace program to make sure whether the packets have been sent or not, by checking the COMPLETION ring structure. In summary, users of XSK needs to properly interact with the following four rings:

- FILL ring: for users to fill UMEM addresses to kernel for receiving packets.
- RX ring: for users to access recevied packets.
- TX ring: for users to place packets needed to be sent.
- COMPLETION ring: for users to make sure packets are sent.

Unlike AF_PACKET which is bound to entire netdev, the XSK is more fine-grained. XSK is bound to a specific queue id on a device, so only the traffic/flow sent to the queue id shows up in the XSK.

3. OVS eBPF Datapath

3.1 eBPF Configuration and Maps

The OVS eBPF datapath consists of one ELF-formatted object file which provides the full functionality of an OVS datapath. This object defines a set of maps and a set of eBPF programs which implement a subset of the datapath functionality. To bootstrap, we load the eBPF program into the kernel using `iproute`. One of the programs is marked within the ELF file to indicate that it should be attached to the hook point; the other programs are only executed via tail calls rooted in the eBPF hook point. The ELF file also defines multiple persistent eBPF maps, which are pinned to the BPF filesystem [**?**] for sharing between different eBPF programs

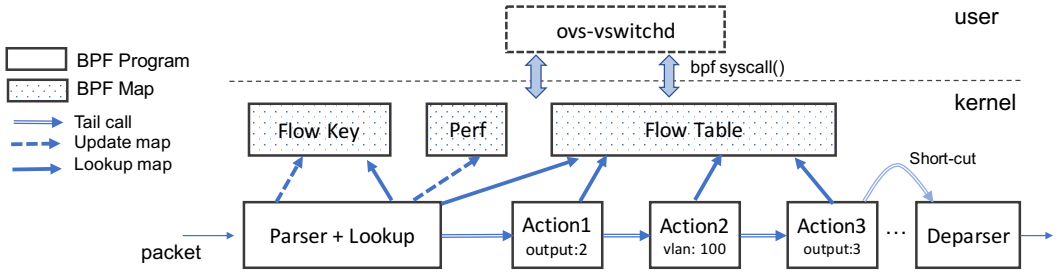

Figure 3: *The overall architecture of OVS eBPF datapath consists of multiple eBPF programs which are tail-called dynamically, maps which are shared between eBPF programs and userspace applications, and ovs-vswitchd as the management plane for all components.*

and `ovs-vswitchd`. The OVS datapath requires the following eBPF maps:

- Flow key. This is the internal representation of the protocol headers and metadata for the current packet being processed, defined in the P4 [**?**] language.
- Flow Table. This is a hash table whose key is the 32-bit hash value of the flow key, from both packet and its metadata, and value equals an array of actions to execute upon the flow.
- Stats Table. This is similar to the flow table, but rather than holding an array of actions to execute for the packet, it contains packet and byte statistics for the flow.
- Perf Ring Buffer [**?** **?**]. The perf event map allows an eBPF program to put user-defined data into a ring buffer which may be read from a userspace program. `ovs-vswitchd` memory maps this ring buffer to read packets and metadata from the eBPF program for flow miss upcall processing.
- Program Array. This map allows eBPF programs to tail call other eBPF programs. When the BPF loader inserts eBPF programs into the kernel, it assigns unique indexes for each program and stores these into the map. At run time, an eBPF program will tail call another program by referring to an index within this map.

The OVS eBPF program is triggered by the TC ingress hook associated with a particular network device. Multiple instances of the same eBPF program may be triggered simultaneously on different cores which are receiving traffic from different network devices. The eBPF maps, unless specified, have a single instance across all cores. Access to map entries are protected by the kernel RCU [**?**] mechanism which makes it safe to read concurrently. However, there are no built-in mechanisms to protect writers to the maps. For the flow table map, OVS avoids the race by ensuring that only `ovs-vswitchd` inserts or removes elements from the map from a single thread. For flow statistics, atomic operations are used to avoid race conditions. Other maps such as the flow key perf ring buffer maps use per-cpu instances to manage synchronization.

3.2 Header Parsing and Lookup

When a packet arrives on the TC ingress hook, the OVS eBPF datapath begins executing a series of programs, beginning with the parser/lookup program as shown in Figure 3. The eBPF parser program consists of two components; standard protocol parsing and Linux-specific metadata parsing. The protocol parsing is executed directly on the packet bytes based on standardized protocols, while the platform-specific metadata parsing must occur on the *context* provided by the eBPF environment.

The resulting code will assemble the protocol headers and metadata, collectively known as the flow key. This flow key is then used to look for an entry in the flow table map, to get an array of actions to execute. If there is no entry in the flow table map, then the packet and the flow key will be written to the perf event map for further processing by `ovs-vswitchd`.

3.3 Action Execution

When a lookup is successful the eBPF gets a list of actions to be executed, such as outputting the packet to a certain port, or pushing a VLAN tag. The list of actions is configured in `ovs-vswitchd` and may be a variable length depending on the desired network processing behaviour. For example, an L2 switch doing unknown broadcast sends packet to all its current ports. The OVS datapath's actions is derived from the OpenFlow action specification and the OVSDB schema for `ovs-vswitchd`.

One might expect to intuitively write an eBPF program to iterate across the list of actions to execute, with each iteration of the loop dealing with a particular action in the list. Unfortunately, this type of iteration implies dynamic loops, which are restricted within the eBPF forwarding model. Moreover, the variable number of actions also implies that there is no way to guarantee the bounded program size, which is limited to 4,096 eBPF bytecode instructions.

To solve these challenges, we first break each type of action logic into an independent eBPF program and tail call from one eBPF action program to another, as shown in Figure 3. This alleviates the problem from having 4k instructions for the *entire* action list to 4k instructions *per action*. Our proof-of-concept implementation shows that this limitation is sufficient for all existing actions. Additionally, the

design allows each action to be implemented and verified independently. Second, to solve the dynamic looping problem, we convert the variable length list of actions into a fixed length 32-element array. As a result, flow table lookup always returns an array of 32 actions to be executed, and the LLVM compiler unrolls the loop to pass the BPF verifier. If a matching flow has less than 32 actions to execute, the rest of the actions is no-op, and we short-cut the action execution to the deparser. If a matching flow has more than 32 actions, then the eBPF datapath delegates the execution to userspace datapath, i.e., the slow path. For many use cases, this is sufficient; there is room to further optimize this path in the future if common cases require more than 32 actions.

Each action also requires certain action-specific metadata to execute. For example, an output action would require an *ifindex* to indicate the output interface, while a push_vlan action needs the *VLAN ID* to be inserted into the VLAN tag. To accommodate this, the array element not only contains the action to execute, but also the metadata required by the particular action. The size of each element must be big enough to hold the metadata for any action, as it is a fixed sized array. Future work may relax this requirement.

Each eBPF program executes in an isolated environment. As a consequence of breaking the action list into individual eBPF action programs, some state needs to be transferred between the current eBPF program and the next. The state to transfer from one eBPF action program to another includes (1) The flow key, so that the currently executing eBPF action program can lookup the flow table, and (2) The action index in the actions list, so that the eBPF action program knows it is executing the Nth element in the list, and at the end of its processing, to tail-call the $n+1$th action program. Currently eBPF could use either per-CPU maps as scratch buffers or the context's packet control block (`cb`) to pass data between programs. In our design, we use one 32-bit value in the `cb` to keep the action index, and per-cpu maps to save the flow key.

Figure 3 also demonstrates an example eBPF execution of a packet forwarded to port 2 as well as port mirroring to a VLAN port with VLAN ID 100 at port 3. Once the packet is parsed, the flow table lookup returns an action list of output:2, push_vlan:100, output:3. At the end of the parser+lookup program, it tail calls the eBPF output program as the first step to kick start action execution. The execution of this output program *overwrites* the caller's stack, so it has to look up the flow table map to retrieve the flow key, and execute the output action. Once done, the output action increments the action index from 0 to 1, saves it in `cb`, and tail-calls the next action program, which is push_vlan. The push_vlan eBPF program again looks up the flow table, and fetches the action metadata at index 1 and executes the push VLAN action using a BPF helper function. The third output action follows the same procedure and finally sends the packet out to port 3.

3.4 Flow Miss Upcall and Installation

One of the important tasks of the OVS datapath is to pass any packet that misses its flow table to the slow path to get instructions for further processing. In the existing OVS *kernel* datapath implementation, the missed packet is sent to `ovs-vswitchd`, which processes the packet, inserts a flow into the flow table, and re-injects the packet into the kernel datapath using the Linux generic netlink interface. For the eBPF datapath, this design implies two requirements: (1) a way for the eBPF program to communicate with userspace, and (2) a mechanism for the userspace to re-insert packet into the eBPF program.

To address the first requirement, we use the support for Linux perf ring buffers and the `skb_event_output()` eBPF helper function [**?**] which allows the eBPF program to pass data to the userspace through the Linux perf ring buffer [**?**]. During miss upcall processing, the eBPF program will insert the full packet and the current flow key into the perf ring buffer. To receive the data from userspace, `ovs-vswitchd` runs a thread which maps the ring buffer to its virtual memory using mmap system call, and polls the buffer using the poll system call. If there is incoming data from the ring buffer, the thread is woken up, it reads from the buffer and processes the packet and metadata. The result of this processing will be inserted into the flow table map. To address the second requirement, we construct a dedicated Linux TAP device which also has the OVS eBPF datapath program attached to it. `ovs-vswitchd` sends the missed packet using an AF_PACKET socket, triggering the underlying eBPF program for further processing. This program is very similar to the previously-used parser+lookup program, with minor changes. Specifically, this packet was originally received on one device, however when `ovs-vswitchd` sends the packet on the TAP device, the eBPF program is triggered for the TAP device instead. So, the platform metadata for incoming port misidentifies the source as the TAP device. To ensure that the packet lookup occurs correctly, `ovs-vswitchd` prepends the port number to the beginning of the packet data, then when the eBPF program for the dedicated TAP device processes the packet, it will read this port number into the metadata, then strip this port from the packet. The resulting packet is identical to the originally-received packet, and now the metadata will match the metadata originally generated by the parser+lookup the first time the packet was received. The rest of the lookup, actions execution, and deparser is then executed as per the description in the previous sections.

3.5 Evaluation

To quantify the performance of the OVS eBPF datapath, we measure the packet forwarding rate in millions of packets per second (Mpps), using 64-byte packets under different forwarding scenarios. The hardware testbed consists of two Intel Xeon E5 2650 servers, each with an Intel 10GbE X540-

Linux Bridge	OVS Kernel	Linux TC	OVS eBPF
1.6 Mpps	1.4 Mpps	1.9 Mpps	1.12 Mpps

Table 2: Baseline port-to-port forwarding rate using existing Linux subsystems and OVS kernel and eBPF.

AT2 dual port NIC, with the two ports of the Intel NIC on one server connected to the two ports on the identical NIC on the other server. The OVS eBPF datapath is installed on one server, acting as a bridge to forward packets from one port on the NIC to the other, and vice-versa. The other server acts as packet generating host, which runs DPDK packet-gen sending at the maximum packet rate of 14.88Mpps. This server sends to one port to the target server, and receives the forwarded packets on the other port. All experiments use only one CPU core running Linux kernel 4.9-rc6.

Baseline Forwarding Performance. We start by conducting a simple port-to-port packet forwarding experiment, i.e., receiving packets from one port and outputting to the other, using the Linux native bridge, Linux TC, and OVS kernel datapath, as shown in Table 2. The native Linux bridge is a simple L2 mac-learning switch with no programmability, showing the forwarding rate of 1.6 Mpps. The OVS kernel module, which does additional flow key extraction, shows a slower forwarding rate of 1.4 Mpps. For forwarding packets using TC, we have an eBPF program loaded into TC, with the program only having one BPF helper function call, the bpf_skb_redirect, that redirects the incoming packet from one interface to another. Since TC accesses the incoming packets closest to the driver layer, it shows the highest performance of 1.9 Mpps. Finally, we measure our proposed OVS eBPF forwarding speed, with incoming packets traversing through an eBPF parser+lookup program, an output action program, and the deparser program. Since the OVS eBPF is implemented based on tail calling these additional eBPF programs at TC, we observe the overhead of 0.78 Mpps, a reduction from 1.9 Mpps to 1.12 Mpps.

Forwarding with action execution. To further investigate the overhead, we program one additional action executed before the packet is forwarded to the other port. Since the Linux bridge has no programmability, we only compare the OVS kernel datapath with the OVS eBPF datapath. Table 3 shows forwarding packets while executing the additional action type: hash, push_vlan, set_dst_mac, and set_gre_tunnel, with the OVS kernel and eBPF datapaths. The forwarding performance of these results is bounded by the CPU cost. In the case of the hash and vlan actions, the NIC allows the processing to be offloaded from the CPU, resulting in minimal performance overhead for executing these actions. As a result, the forwarding rate exhibits little to no overhead above the baseline. Moreover, for the hash and push_vlan actions, due to the NIC offloading of actions processing, the datapath does not need to execute any modification to the packet contents, so the deparser program execution may be optimized out.

Action	eBPF DP	Kernel DP	Overhead
hash	1.12	1.34	16%
push_vlan	1.11	1.32	15%
set_dst_mac	0.84	1.28	34%
set_gre_tunnel	0.48	0.57	8%

Table 3: Comparison of single core forwarding rate in Mpps with eBPF and Kernel Datapath, with the additional action executed before forwarding the packet to another port.

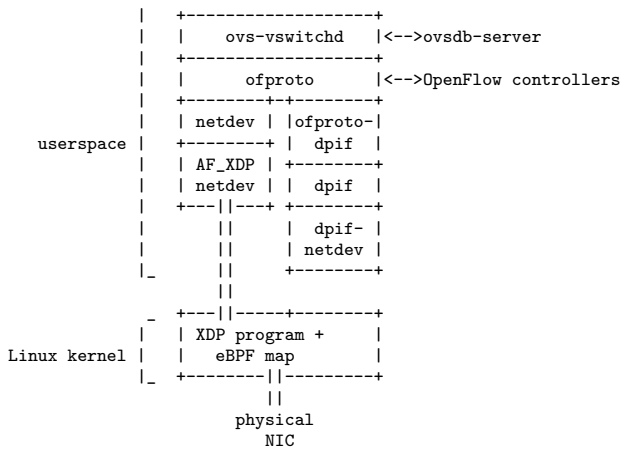

Figure 4: OVS Architecture AF_XDP

To ensure invoking the deparser, the third experiment, set_dst_mac, involves altering packet data, specifically to modify the destination MAC address. In this case, the deparser writes stored metadata from actions processing back to the packet. The performance in this experiment drops to 0.84 Mpps. The drop of 0.28 Mpps from baseline is the result of the deparser writing all known packet header metadata back to the packet, and due to extraneous memory copies introduced by the P4-to-eBPF compiler. With a more intelligent deparser implementation, and by using recent improvements to the eBPF API [**? ?**], the performance gap compared with the native kernel implementation is expected to shrink.

Finally, the set_get_tunnel action experiment yielded 0.48 Mpps, which represents the additional cost of tunnelling traffic; such traffic must traverse the Linux network stack twice, once for overlay traffic and once for underlay traffic. The eBPF result is also more expensive than the existing Linux implementation, but with a narrower performance gap than the earlier experiments. The majority of processing in this case occurs outside of OVS, so the overhead of the eBPF datapath has less effect.

4. Userspace OVS with AF_XDP

4.1 Datapath and Netdev Interface

OVS includes a userspace datapath interface (dpif) implemenation, called dpif-netdev. Unlike the dpif-netlink which talks to the Linux Open vSwitch kernel module through netlink API, the dpif-netdev works completely in userspace. As a result, dpif-netdev expects to receive and send packets from its userspace interface. One major use case of dpif-netdev is OVS-DPDK [**?**], where the packet reception and transmission are all done in DPDK's userspace library. The dpif-netdev is designed to be agnostic to how the network device accessing the packets, by an abstraction layer called netdev. So Packets might come from DPDK packet I/O library, a Linux AF_PACKET socket API, or in our case, the AF_XDP socket interface, as long as each mechanism implements its own netdev interface. Once dpif-netdev receives a packet, it follows the same mechanism doing parse, lookup the flow table, and apply actions to the packet.

Figure 4 shows the architecture of userspace OVS with AF_XDP. We implement a new netdev type for AF_XDP, which receives and transmits packets using the XSK. We insert a XDP program and a eBPF map with interact 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.

4.2 AF_XDP netdev configuration

When users attach a AF_XDP netdev to an OVS bridge, for example, by issuing the following commands:

```
ovs-vsctl add-br br0
ovs-vsctl add-port br0 eth0 -- \
    set int eth0 type="afxdp"
```

ovs-vswitch does the following steps to bring up the AF_XDP netdev:

1. Attach a XDP program to the netdev's queue: The XDP program OVS attaches is fixed and consists of only a few line of code, which receives the packet and calls the `bpf_redirect_map()` helper function with the XSK eBPF maps and key equals zero.
2. Create a AF_XDP socket: Call socket() syscall to create a XSK, set up its RX and TX ring buffer, allocate a UMEM region, and set up FILL/COMPLETION ring of the UMEM.
3. Load and configure the XSK eBPF map: The XSK eBPF maps consists of key value pairs, where key is an u32 index and value is the file descript of the XSK. ovs-vswitchd simply programs one entry to the map with key equals zero and the file descriptor, fd, of the XSK as its value. As a result, the XDP program calling `bpf_redirect_map` with key=0 will forwards the packet to the XSK.

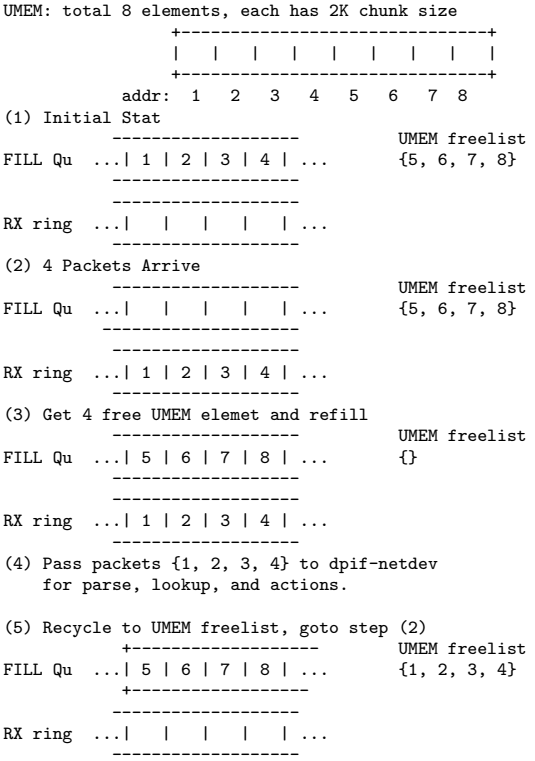

Figure 5: An example of how OVS programs the FILL ring and RX ring when processing incoming AF_XDP packets.

4. Populate the UMEM FILL ring: Get a couple UMEM elements and place into FILL ring.

On the other hand, when a AF_XDP netdev is detached or closed by user, ovs-vswitchd closes the XSK socket, free the UMEM memory region, and unload the eBPF program and map.

In order to properly set-up the RX/TX/FILL/COMPLETION rings, we implement a UMEM memory management API. Basically, it's a list maintaining the unused elements in UMEM, called UMEM freelist, and function calls to get/put elements into the list.

4.3 Packet Reception

Figure 5 shows how ovs-vswitchd sets up the FILL ring and the RX ring for receving packets from XSK. For simplicity to demonstrate how AF_XDP works, assume that there are only eight UMEM buffers, with each buffer's size equals 2KB. Initially at step 1, ovs-vswitchd fills four empty UMEM elem into the FILL ring and waits for incoming packets. When there are incoming packets, at step 2, the FILL ring's four buffer elems have been consumed and moved to the RX ring. In order to keep receiving packets, ovs-vswitchd gets another four empty UMEM elems from thte UMEM freelist, and fills into FILL ring (step 3). Then ovs-vswitchd creates the metadata needed for the four packet buffer, i.e., struct dp_packet and struct dp_packet_batch, and passes to the dpif-netdev layer for parse, lookup and action

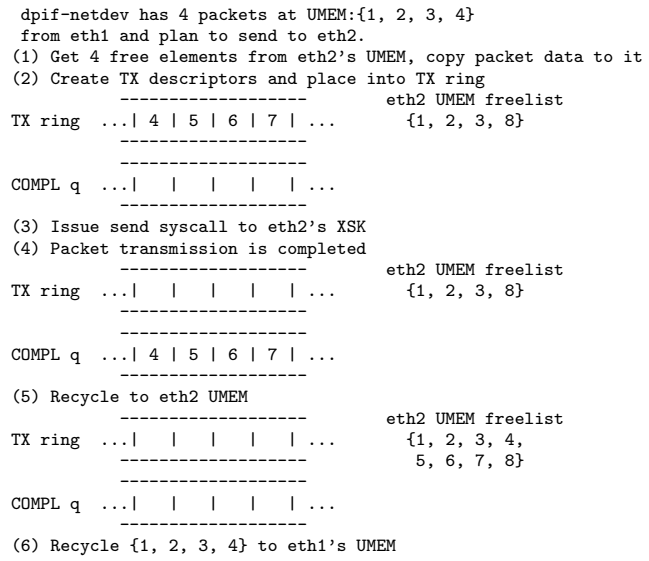

Figure 6: An example of how OVS programs the COMPLETION ring and TX ring when processing incoming AF_XDP packets from one netdev and sending to another netdev.

executions. Finally, when ovs-vswitchd finishes processing the UMEM buffer, a recycle mechanism is triggered to place these buffer back to UMEM freelist (step 5). Step 5 makes sure that there are always available elems in FILL ring, so that the underlying XDP program in kernel can keep placing packets into the FILL ring while the userspace ovs-vswitchd is processing the previous received packets. When step 5 finishes, ovs-vswitchd goes back to step 2, waiting and processing new packets.

4.4 Packet Transmission

Figure 6 shows the process for sending packets to the XSK. Assuming that ovs-vswitch has a a bridge and two ports: eth1 and eth2. And there is a flow entry showing `in_port=eth1, ..., actions=..., output:eth2`, meaning to forwarding packets received from eth1 to eth2. And both eth1 and eth2 are configured with AF_XDP support.

Initially, assume ovs-vswitchd receives four packets 1, 2, 3, 4 from eth1's XSK To send to eth2 using its XSK, ovs-vswitchd first gets four packet buffers, 4, 5, 6, 7, from eth2's UMEM freelist and copy the packet data to their packet buffer in UMEM. Then TX descriptors for the four packets are created and placed into eth2's TX ring (step 2). At step 3, send syscall is issued to signal the kernel to start transmit. As the send in XSK is asynchronos and send syscall only return return zero when no error, ovs-vswitch polls the COMPLETION ring to make sure these four packets have been sent out (step 4). Once the four packets' descriptors shown up at the COMPLETION ring, at step 5, ovs-vswitchd recycles their UMEM elememnts back to the eth2's UMEM freelist. In addition, the original four packet buffer from eth1, 1, 2, 3, 4, is also srecycled back to eth1's UMEM freelist.

4.5 Performance Evaluation

All of our performance results use a hardware test bed that consists of two Intel Xeon E5 2440 v2 1.9GHz servers, each with 1 CPU socket and 8 physical cores with hyperthreading enabled. One server, the target server, has an Intel 40GbE XL710 single port NIC, and the other server, the source server, has Netronome NFP-4000 40GbE device. The target server runs the Linux kernel 4.19-rc4, with i40e driver supporting the AF_XDP mode. We installed OVS-AFXDP on the target server, and the other server, the source server, generates 64-byte single UDP flow packets at the rate of 19 Mpps using the DPDK library. On the target server, we also disable the spectre and meltdown fixes.

For all our experiments, we fuse a microbenchmark program, called xdpsock, as the baseline to compare with OVS-AFXDP implementation. xdpsock is an AF_XDP sample program doing no packet processing, and can be configured to do simply dropping all packets, rxdrop, or forwarding the packet to the same port as it receives, l2fwd. In our testbed, we measured $19Mpps$ for xdpsock-rxdrop and $17Mpps$ for xdpsock-l2fwd. For OVS-AFXDP, we conduct similar two experiments, but with OVS's OpenFlow rule installed as below:

- OVS-AFXDP rxdrop: Install a single OpenFlow rule to drop every packets, e.g., `in_port=eth1, actions=drop`.
- OVS-AFXDP l2fwd: Install a single OpenFlow rule to forward packet to the same port as it receives, e.g., `in_port=eth1, actions=set_fields:eth2->in_port, output:eth1`.

Additionally, AF_XDP provides three operation modes: XDP_SKB: works on devices using generic XDP [**?**], XDP_DRV: works on devices with XDP support, and XDP_DRV ZC (Zero Copy): works on devices with XDP_XDP zerecopy support. For all our experiments, we use the XDP_DRV ZC mode.

4.5.1 PMD netdev

Our initial prototype shows pretty terrible performance, only $0.5Mpps$ for rxdrop. We observe that when dropping/forwarding packets under OVS-AFXDP, two CPUs show up 100% utilitization:

- ovs-vswitchd: This is the process keeps doing the send and receive steps in Figure 5 and 6.
- ksoftirqd/core_id: This is the kernel software interrupt thread handling the incoming packets, triggering XDP program to pass packets to the XSK, and also processing transmission.

We investigate using Linux perf and our first fix is to enable OVS's PMD-mode, Poll-Mode-Driver, to the afxdp netdev.

In OVS, a non-pmd mode netdev does packet receiption by putting all the receiving netdev's file descriptors, fds, together and ovs-vswitchd polls them when there is an fd

ready to perform I/O. As a result, the fd of the XSK is *shared* with other fds, and we observed that the poll system call has pretty high overhead.

Applying OVS's PMD netdev avoids these problems and improving the rxdrop performance from $0.5Mpss$ to $3Mpps$. Ideally, receing packets from XSK should not need to context switch to into kernel as mentioned in steps 5. So an ideal implementation should show minimal context switches with most of the CPU time stays in userspace. When enabling PMD netdev for afxdp, ovs-vswitch can be configured to create a *dedicated* thread for the packet processing. In OVS-AFXDP, we create a thread for each XSK's receive queue and the thread keeps polling the RX ring for new packets. Each round of receive polling processes a batch of packet, up to maximum of 32 packets. In addition, we enable 1GB huge memory page support to reduce the page fault overhead. With the above fixes/optimizations, the rxdrop performance increases to around $10Mpps$

4.5.2 Memory Management and Locks

For every stage of the optimization, we use Linux perf extensively, e.g., perf stat and perf record/report. With the above new design, we observe the new bottleneck is at the UMEM memory pool APIs we introduce to maintain the UMEM freelist, as well as the packet metadata allocation, struct dp_packet, for every receiving packet. We introduce two major API, umem_elem_get(), which gets N free element from the UMEM freelist, and umem_elem_put(), which places back the free UMEM buffer to the freelist, similar to the concept of producer (put) and consumer (get). We implemented three different data structures of UMEM memory pool as below to compare their performance.

- list_head: Similar to Linux kernel's list_head, each element contains its UMEM element address, and a next pointer points to next free element. Due to the nature of list_head, elements are sparse in memory. Consumer always takes the first element, head, and producer also place back from the head of the list.
- ptr_ring: Similar to Linux kernel's ptr_ring, an array of elements are allocated in continuous memory region. Consumer takes elements from the tail pointer and when placing back, producer, puts them into where head pointer points to.
- ptr_stack: Similar to ptr_ring, an array of elements are allocated in continuous memory region, but both consumer and producer takes elements from beginning of the array.

Although we allocate one UMEM per netdev, there might be multiple queues per netdev sharing the same UMEM. As a result, the above three data structures require a mutex lock before accessing UMEM freelist. Linux perf reports pthread mutex lock related APIs as one of the top CPU utilization function. We change our design by 1) allocating per-queue UMEM region and 2) allocating one pmd thread per queue. As a result, no lock is needed because the each queue has only one thread and its own set of UMEM elements.

4.5.3 Metadata Pre-allocation

Moving forward, Linux perf shows that the packets metadata allocation takes a lot of CPU cycles, i.e., dp_packet_init(), dp_packet_set_data(). So instead of allocating packet metadata at packet reception time, we implemented two data structures and compare their performance:

- Embedding in packet buffer: Since we already allocate 2K chunk for each UMEM packet buffer, we reserve the first 256-byte in each UMEM element as struct dp_packet and initialize the dp_packet as much as we can at allocation time. This is similar to the DPDK mbuf design [**?**], where a single memory buffer is used for both packet data and metadata.
- Separate from packet buffer: This design simply allocate a continuous memory region storing an array of packet metadata, and initialize them as much as we can.

With the above design change, we found that using the ptr_stack in section 4.5.2 with separate from packet buffer design yields the best performance. Largely due to both data structures have better spatial locality and more batching friendly, when all data are allocated together. For example, setting up 32 packet metadatas in an array definitely incurs less cache misses than setting up in 32 elements in UMEM. And getting 32 free UMEM elements from ptr_stack has similar benefits. With the above decision, the OVS rxdrop shows around $19Mpps$, closed to the baseline xdpsock rxdrop performance.

4.5.4 Batching Send Syscall

We continued measuring the performance of OVS-AFXDP l2fwd and observed only $4Mpps$, compared to $17Mpps$ baseline xdpsock-l2fwd. We found that the OVS pmd thread under rxdrop has much fewer context switches compared to the l2fwd, indicating that the pmd process spends much more time in kernel space than in userspace. By using strace, we saw that the OVS-AFXDP l2fwd experiement calls sendto system call at very high frequeuce, because from Figure 6, we design to check the success of send (step 4) immediately after issusing send (step 3). We change the design by calling send syscall (step 3) only when TX ring is closed to full, e.g., when 3/4 ring elements have been used. In specific case, instead of issuing send syscall for a batch of 32 packets and making sure they are done, we only issue send when there are 512 outstanding packets. With this change, the OVS-AFXDP l2fwd experiment shows around $14Mpps$.

4.5.5 Summary

In summary, with the above design and optimization, OVS-AFXDP shows Although there are still rooms for improvement, we are now working on making the patch upstream to the OVS code base for more people to use.

Projects	xdpsock	OVS-AFXDP
rxdrop	19Mpps	19Mpps
l2fwd	17Mpps	14Mpps

Table 4: Performance comparison of the xdpsock and and OVS-AFXDP.

5. Conclusion

References

[1] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, volume 46, 1993.