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. On the other hand, the OVS-AFXDP project aims to by-passing the kernel using AF_XDP socket and moving most of the flow processing features in userspace.

1. Introduction

Among the various ways of using eBPF, OVS has been exploring the power of eBPF in three: (1) attaching eBPF to TC, (2) offloading a subset of processing to XDP, and (3) by-passing the kernel using AF_XDP. Attaching eBPF to TC 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 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, we are working on connection tracking. Although a simple eBPF map can achieve basic operations of conntrack table lookup and commit, how to handle NAT, (de)fragmentation, and ALG are still under discussion.

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

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

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

```
                +-------------------+
 slow path      |    ovs-vswitchd   |
                +-----^-------------+
                      |       |flow installation
----------------------|-------|-----------------
 fast path     upcall |       |
 (datapath)  +----------------v---------+
  packets--> | parse, lookup, actions   |  --> output
             +--------------------------+
```

Figure 1: An example of mirrored packet with outer header containing the GRE and ERSPAN header, followed by the innter Ethernet frame.

flows are eligible for offload, or 2) more flow processing logic needed to be done in native eBPF.

AF_XDP is another form of XDP but providing a 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, used by DPDK community. By treating the AF_XDP as a fast packet-I/O channel, the OVS dpif-netdev can satisfy most of the features. We are working on building the prototype and evaluating its performance.

Existing OVS consists of three datapath implementations: dpif-netlink dpif-netdev dpif-windows In this paper, OVS-eBPF project adds a new datapath type dpif-bpf and OVS-AFXDP project reuses the dpif-netdev interface, with linux netdev receiving/sending packets from AFXDP sockets.

Table 1 shows the difference between two projects:

2. Background

2.1 OVS Forwarding Model

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 **??**, 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 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.

2.2 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 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 instruc-

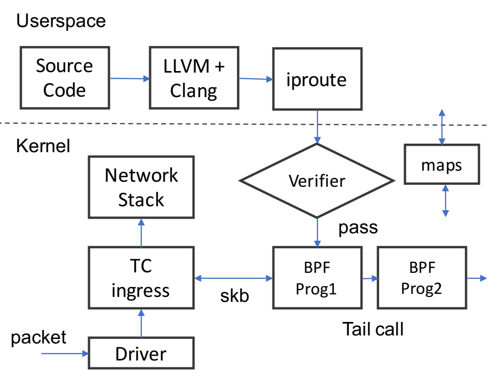

Figure 2: *The workflow of TC and XDP eBPF development process and its packet flow. The eBPF program compiled by LLVM+clang is loaded into the kernel using iproute. The kernel runs the program through a verification stage, and subsequently attaches the program to the TC/XDP ingress hook point. Once successfully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

tions. 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

```
             |   +-------------------+
             |   |    ovs-vswitchd   |<-->ovsdb-server
             |   +-------------------+
             |   |      ofproto      |<-->OpenFlow controllers
             |   +--------+-+--------+
             |   | netdev | |ofproto-|
   userspace |   +--------+ |  dpif  |
             |   | AF_XDP | +--------+
             |   | netdev | |  dpif  |
             |   +---||---+ +--------+
             |       ||     |  dpif- |
             |       ||     | netdev |
             |_      ||     +--------+
                     ||
              _  +---||-----+--------+
             |   | XDP program +     |
Linux kernel |   |   eBPF map        |
             |_  +--------||---------+
                          ||
                       physical
                          NIC
```

Figure 3: OVS Architecture AF_XDP

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

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 3 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 the OVS bridge, for example, by issuing the following command:

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

ovs-vswitch does the following steps to configure 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.
4. Fill 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.

```
UMEM: total 8 elements, each has 2K chunk size
                 +-------------------------------+
                 |   |   |   |   |   |   |   |   |
                 +-------------------------------+
            addr:  1   2   3   4   5   6   7  8
(1) Initial Stat
           -------------------         UMEM freelist
FILL Qu  ...| 1 | 2 | 3 | 4 | ...      {5, 6, 7, 8}
           -------------------
           -------------------
RX ring  ...|   |   |   |   | ...
           -------------------
(2) 4 Packets Arrive
           -------------------         UMEM freelist
FILL Qu  ...|   |   |   |   | ...      {5, 6, 7, 8}
          --------------------
           -------------------
RX ring  ...| 1 | 2 | 3 | 4 | ...
           -------------------
(3) Get 4 free UMEM elemet and refill
           -------------------         UMEM freelist
FILL Qu  ...| 5 | 6 | 7 | 8 | ...      {}
           -------------------
           -------------------
RX ring  ...| 1 | 2 | 3 | 4 | ...
           -------------------
(4) Pass packets {1, 2, 3, 4} to dpif-netdev
    for parse, lookup, and actions.

(5) Recycle to UMEM freelist, goto step (2)
            +-------------------       UMEM freelist
FILL Qu  ...| 5 | 6 | 7 | 8 | ...      {1, 2, 3, 4}
            +------------------
           -------------------
RX ring  ...|   |   |   |   | ...
           -------------------
```

Figure 4: An example of how OVS programs the FILL ring and RX ring when processing incoming AF_XDP packets.

4.3 Packet Reception

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 APIs to get/put elements into the list.

Figure 4 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 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

```
 dpif-netdev has 4 packets at UMEM:{1, 2, 3, 4}
 from eth1 and plan to send to eth2.
(1) Get 4 free elements from eth2's UMEM, copy packet data to it
(2) Create TX descriptors and place into TX ring
           -------------------         eth2 UMEM freelist
TX ring  ...| 4 | 5 | 6 | 7 | ...        {1, 2, 3, 8}
           -------------------
           -------------------
COMPL q  ...|   |   |   |   | ...
           -------------------
(3) Issue send syscall to eth2's XSK
(4) Packet transmission is completed
           -------------------         eth2 UMEM freelist
TX ring  ...|   |   |   |   | ...        {1, 2, 3, 8}
           -------------------
           -------------------
COMPL q  ...| 4 | 5 | 6 | 7 | ...
           -------------------
(5) Recycle to eth2 UMEM
           -------------------         eth2 UMEM freelist
TX ring  ...|   |   |   |   | ...        {1, 2, 3, 4,
           -------------------          5, 6, 7, 8}
           -------------------
COMPL q  ...|   |   |   |   | ...
           -------------------
(6) Recycle {1, 2, 3, 4} to eth1's UMEM
```

Figure 5: 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.

finishes, ovs-vswitchd goes back to step 2, waiting and processing new packets.

4.4 Packet Transmission

Figure 5 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, 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.

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 the packet, 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 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: 1) XDP_SKB: works on devices using generic XDP [**?**], 2) XDP_DRV: works on devices with XDP support. 3) XDP_DRV ZC (Zero Copy): works on devices with XDP_XDP zerecopy support. For all our experiments, we use the XDP_DRV ZC mode.

When forwarding packets using AF_XDP with OVS, two CPUs show up 100% utilitization:

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

Compared to xdpsock, ovs-vswitchd adds on top of it a UMEM memory pool, and packets parsing, lookup and action execution. Our initial prototype shows pretty terrible performance, only $0.5Mpps$ rxdrop. We investigated and our first fix is to enable OVS's PMD-mode, Poll-Mode-Driver, to the afxdp netdev.

4.5.1 PMD netdev

In OVS, a non-pmd mode netdev does packet receiption by putting all the receiving netdev's file descript together and ovs-vswitch 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 does not need to context switch to into kernel as mentioned in steps 4.

So an ideal implementation should show minimal context switches with most of the CPU stays in userspace. When enabling PMD netdev for afxdp, ovs-vswitch can be configured to create a *dedicated* thread for the packet I/O. In our case, we create a thread for each XSK receive queue which 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 high memory page 1GB 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 we introduce to maintain the UMEM freelist, and 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. producer consumer We implemented three data structures of UMEM memory pool as below to compare their performance

- list_head: Similar to Linux kernel's list_head, each element contains it UMEM addr, and a next pointer points to next free element. 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, and pointers (head and tail) point for Consumer takes elements from the tail 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 all require a mutex lock before accessing UMEM freelist. Linux perf reports pthread_mutex_lock as one of the top CPU utilization function, and overall shows around $2Mpps$ overhead for rxdrop experiments. 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.

With the above change, Linux perf shows that the packets metadata allocation takes a lot of CPU cycles, the dp_packet_init, dp_packet_set_data. So instead of allocating packet metadata at packet reception time, we implement two data structures and compare their performance:

- Inlining in UMEM buffer: Since we already allocate 2K chunk for each UMEM 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. So when packets arrive, each packet already has its packet metadata at a const offset ahead.
- Array on heap: 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 with array on heap get the best performance. Largely due to both data structures have better spatial locality and more batching friendly. 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 $17Mpps$, only $2Mpps$ less than the baseline xdpsock rxdrop.

4.5.3 Batching Send Syscall

We continued measured the performance of OVS-AFXDP l2fwd and. observed only $4Mpps$, compared to $17Mpps$ baseline xdpsock. Again, by analysis using perf, 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 l2fwd experiement calls sendto system call at very high frequeuce, because from Figure 5, 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 $14Mpps$.

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.