Bringing the Power of eBPF to Open vSwitch

Linux Plumber 2018

William Tu, Joe Stringer, Yifeng Sun, Yi-Hung Wei

VMware Inc. and Cilium.io

Outline

- Introduction and Motivation
- OVS-eBPF Project
- OVS-AF_XDP Project
- Conclusion

What is OVS?

SDN Controller

OpenFlow

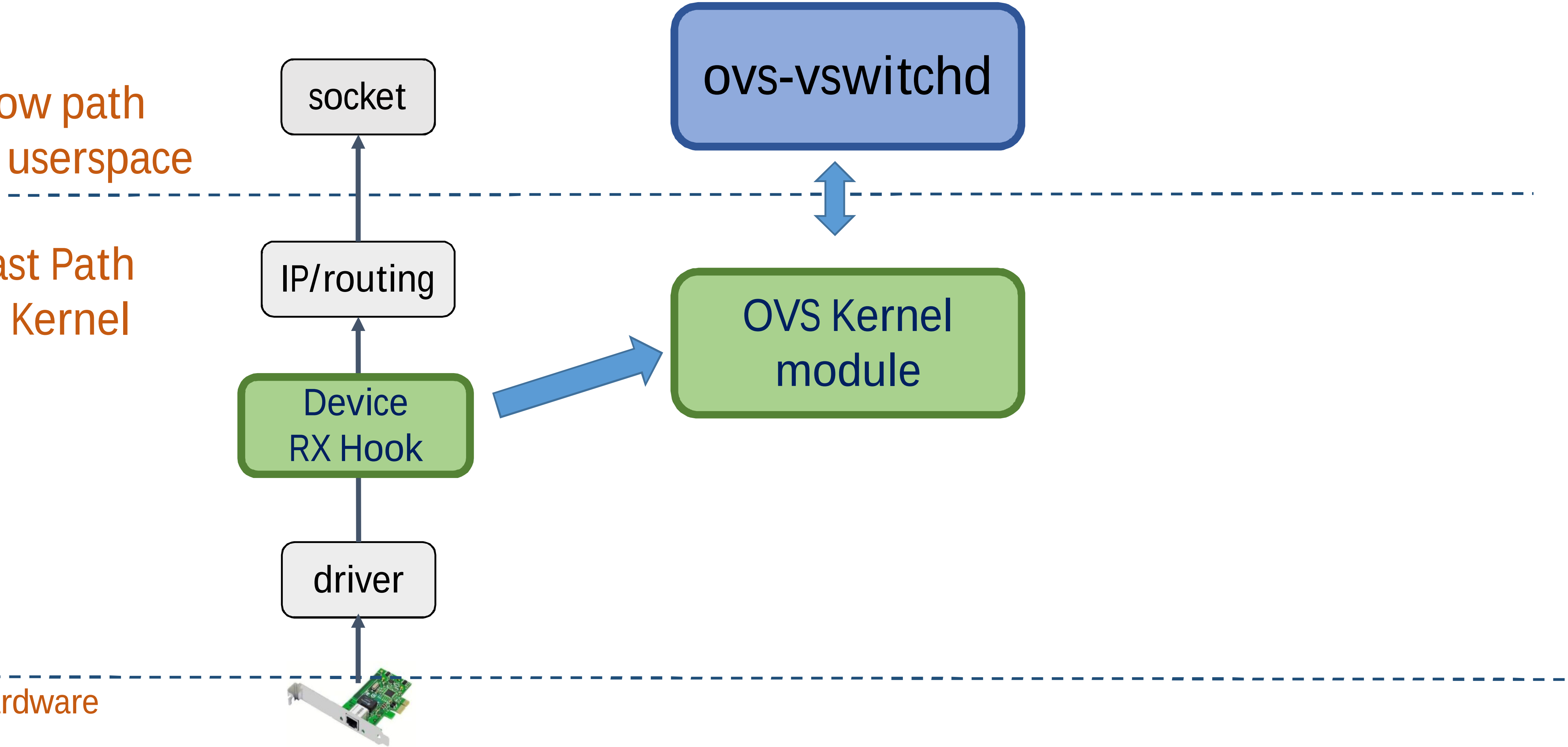
ovs-vswitchd

Slow Path

Fast Path

Datapath

OVS Linux Kernel Datapath

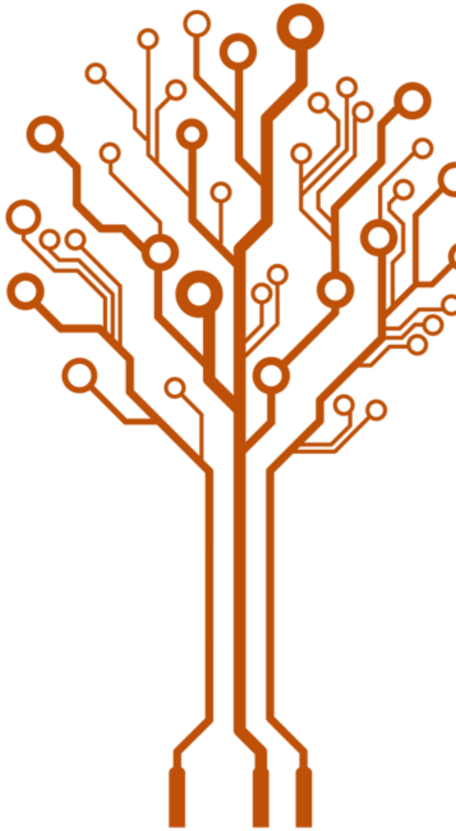


OVS-eBPF

OVS-eBPF Motivation

- Maintenance cost when adding a new datapath feature:
  - Time to upstream and time to backport
  - Maintain ABI compatibility between different kernel and OVS versions.
  - Different backported kernel, ex: RHEL, grsecurity patch
  - Bugs in compat code are easy to introduce and often non-obvious to fix

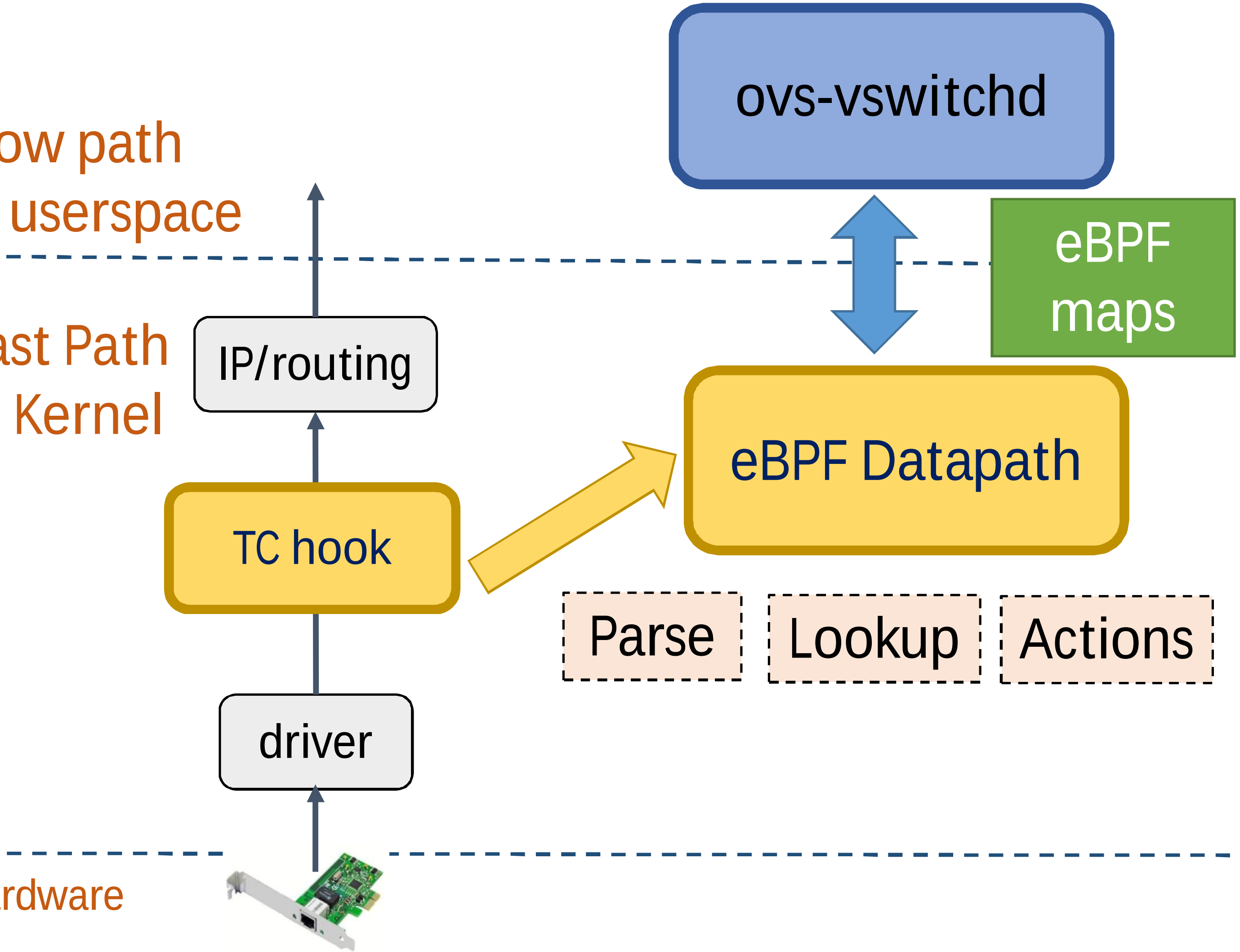
- Implement datapath functionalities in eBPF
  - Reduce dependencies on different kernel versions
  - More opportunities for experiements



What is eBPF?

- A way to write a restricted C program and runs in Linux kernel
  - A virtual machine running in Linux kernel
  - Safety guaranteed by BPF verifier
- Maps
  - Efficient key/value store resides in kernel space
  - Can be shared between eBPF prorgam and user space applications
- Helper Functions
  - A core kernel defined set of functions for eBPF program to retrieve/push data from/to the kernel

OVS-eBPF Project



<u>Goal</u>

- Re-write OVS kernel datapa entirely with eBPF
- ovs-vswitchd controls and manages the eBPF DP
- eBPF map as channels in between
- eBPF DP will be specific to ovs-vswitchd

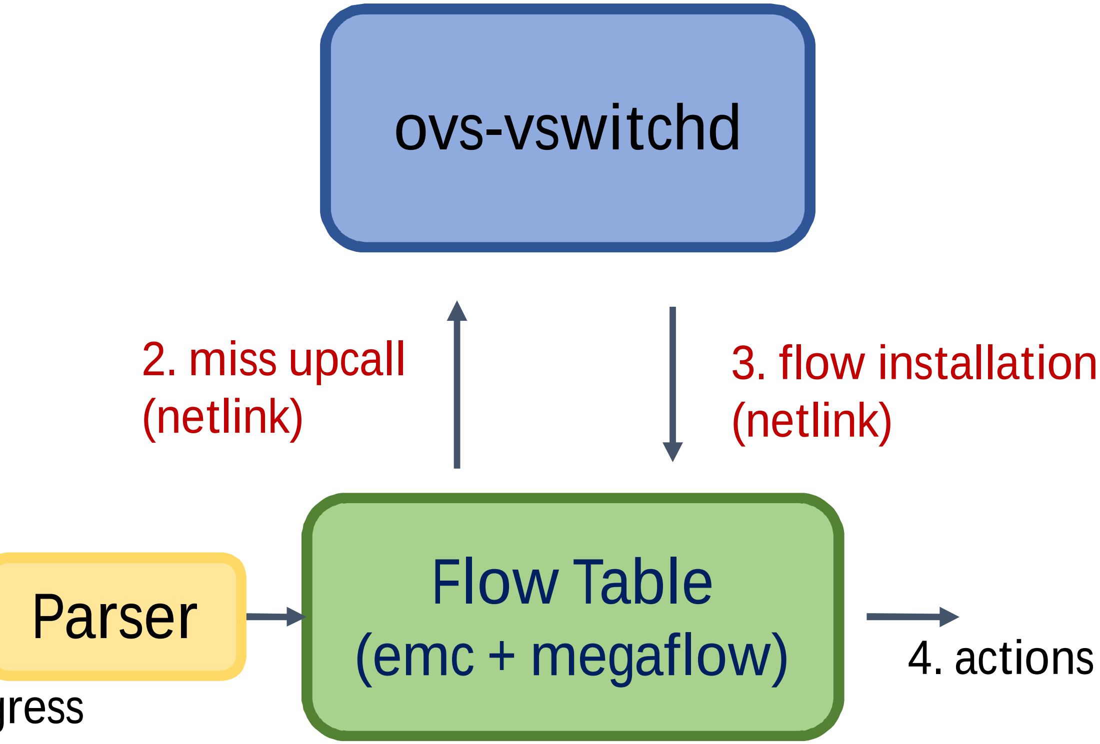
Headers/Metadata Parsing

- Define a flow key similar to struct sw_flow_key in kernel
- Parse protocols on packet data
- Parse metadata on struct __sk_buff
- Save flow key in per-cpu eBPF map

Difficulties

- Stack is heavily used
- Program is very branchy

Review: Flow Lookup in Kernel Datapath



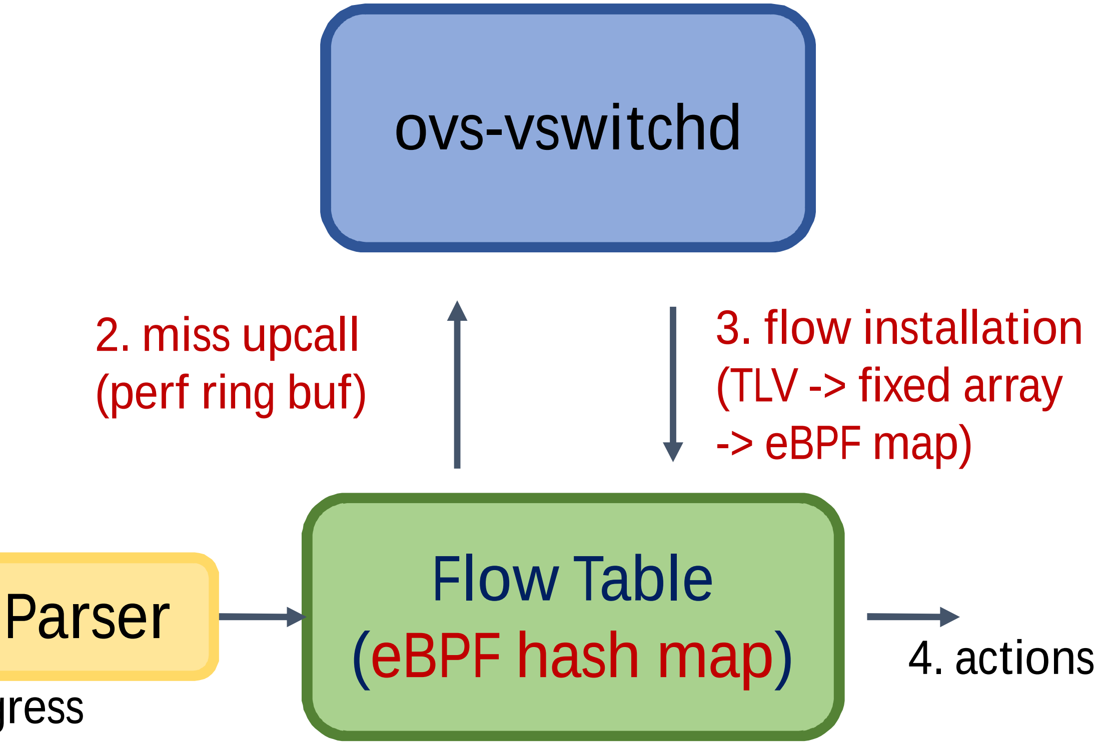
Slow Path

- Ingress: lookup miss and upcall
- ovs-vswitchd receives, does flow translation, and programs flow entry into flow table in OVS kernel module
- OVS kernel DP installs the flow entry
- OVS kernel DP receives and executes actions on the packet

Fast Path

- Subsequent packets hit the flow cache

Flow Lookup in eBPF Datapath



imitation on flow installation:
TLV format currently not supported in BPF verifier
Solution: Convert TLV into fixed length array

Slow Path

- Ingress: lookup miss and upcall
- Perf ring buffer carries packet and its metadata to ovs-vswitchd
- ovs-vswitchd receives, does flow translation, and programs flow entry into eBPF map
- ovs-vswitchd sends the packet down t trigger lookup again

Fast Path

- Subsequent packets hit the flow cach

Review: OVS Kernel Datapath Actions

A list of actions to execute on the packet

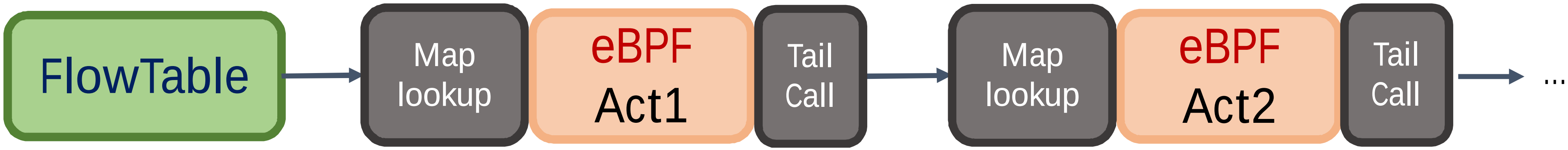
FlowTable → Act1 → Act2 → Act3 → ...

Example cases of DP actions

- Flooding:
  - Datapath actions= output:9,output:5,output:10,...
- Mirror and push vlan:
  - Datapath actions= output:3,push_vlan(vid=17,pcp=0),output:2
- Tunnel:
  - Datapath actions: set(tunnel(tun_id=0x5,src=2.2.2.2,dst=1.1.1.1,ttl=64,flags(df|key))),output:1

eBPF Datapath Actions

A list of actions to execute on the packet



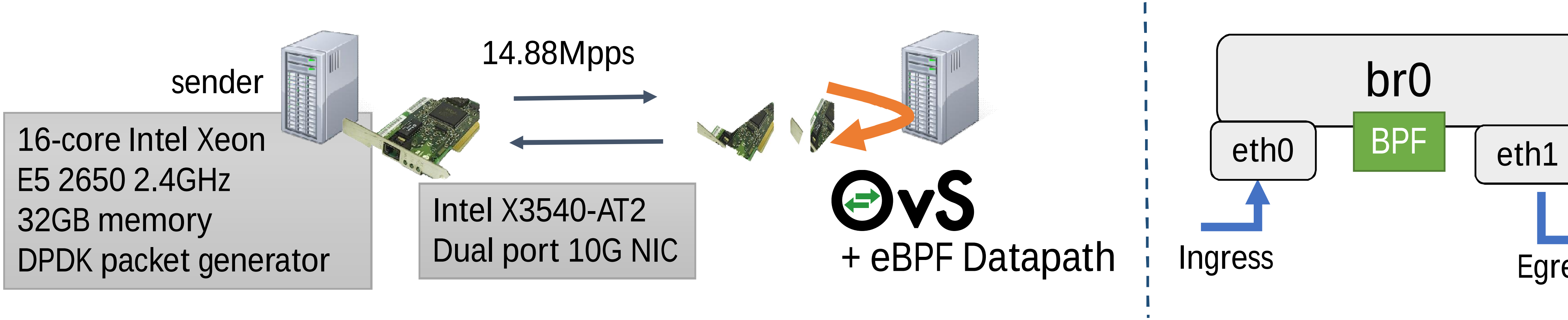
Challenges

- Limited eBPF program size (maximum 4K instructions)
- Variable number of actions: BPF disallows loops to ensure program termination

Solution:

- Make each action type an eBPF program, and tail call the next action
- Side effects: tail call has limited context and does not return
- Solution: keep action metadata and action list in a map

Performance Evaluation



- Sender sends 64Byte, 14.88Mpps to one port, measure the receiving packet rate at the other port
- OVS receives packets from one port, forwards to the other port
- Compare OVS kernel datapath and eBPF datapath
- Measure <u>single flow, single core</u> performance with Linux kernel 4.9-rc3 on OVS server

OVS Kernel and eBPF Datapath Performance

OVS Kernel DP Actions	Mpps
Output	1.34
Set dst_mac	1.23
Set GRE tunnel	0.57

eBPF DP Actions	Mpps
Redirect(no parser, lookup, actions)	1.90
Output	1.12
Set dst_mac	1.14
Set GRE tunnel	0.48

All measurements are based on single flow, single core.

Conclusion and Future Work

Features

- Megaflow support and basic conntrack in progress
- Packet (de)fragmentation and ALG under discussion

Lesson Learned

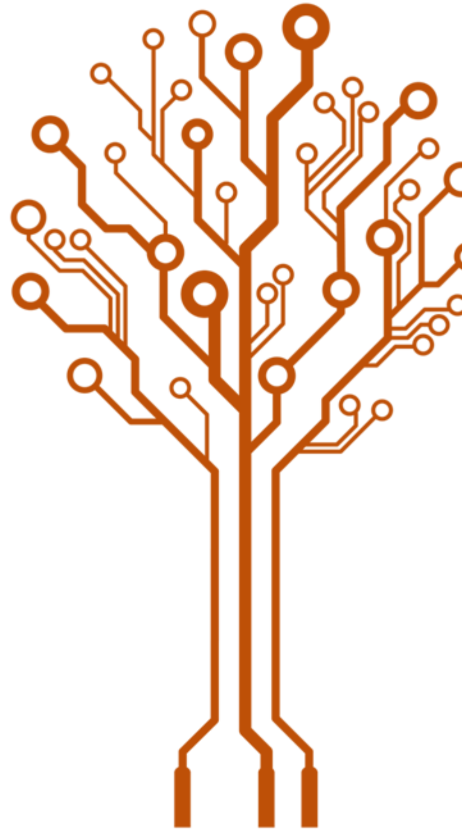
- Writing large eBPF code is still hard for experienced C programmers
- Lack of debugging tools
- OVS datapath logic is difficult

OVS-AF_XDP

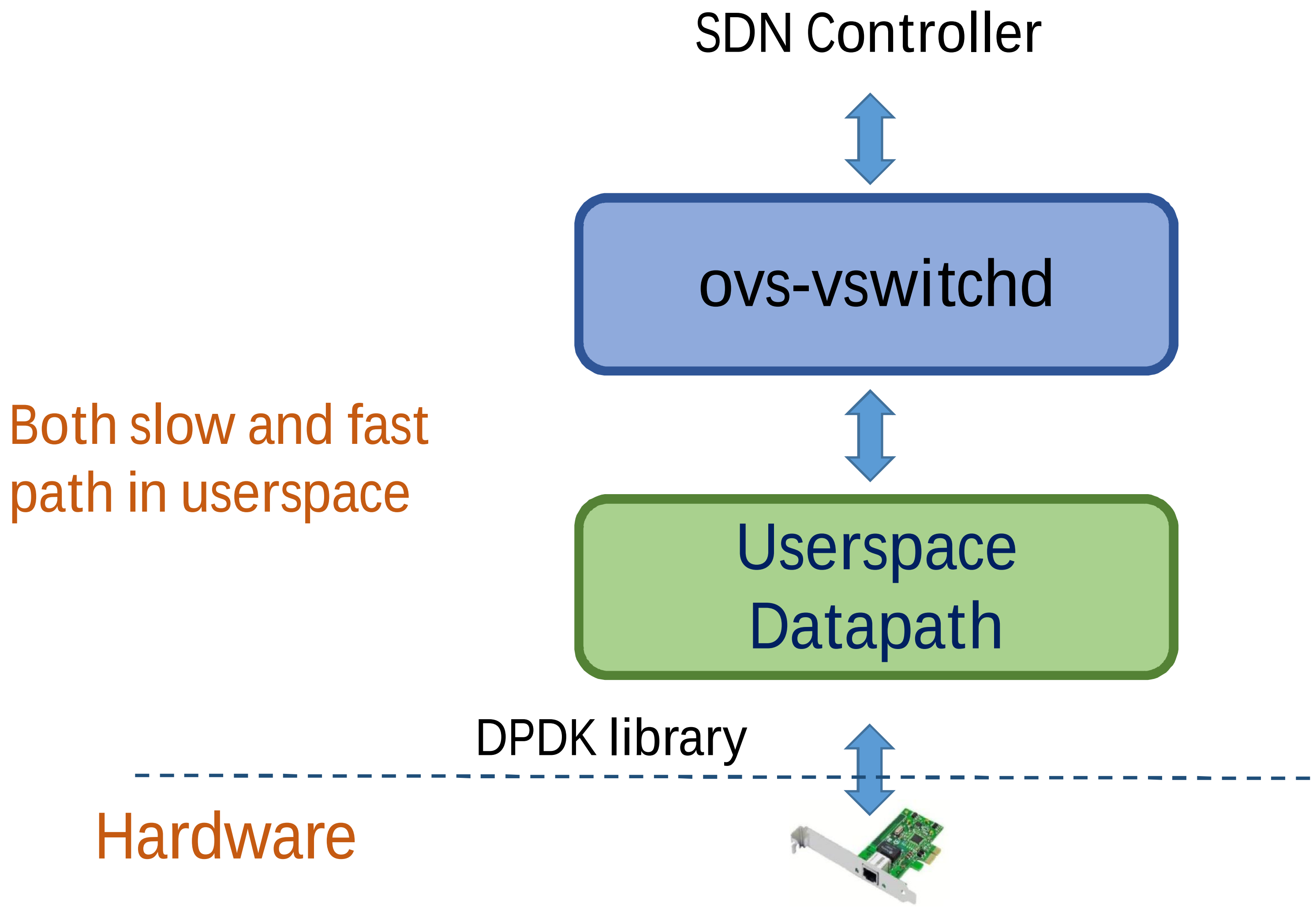
OVS-AF_XDP Motivation

- Pushing all OVS datapath features into eBPF is hard
  - A large flow key on stack
  - Variety of protocols and actions
  - Dynamic number of actions applied for each flow

- Idea
  - Retrieve packets from kernel as fast as possible
  - Reuse the userspace datapath for flow processing
  - Less kernel compatibility than OVS kernel module

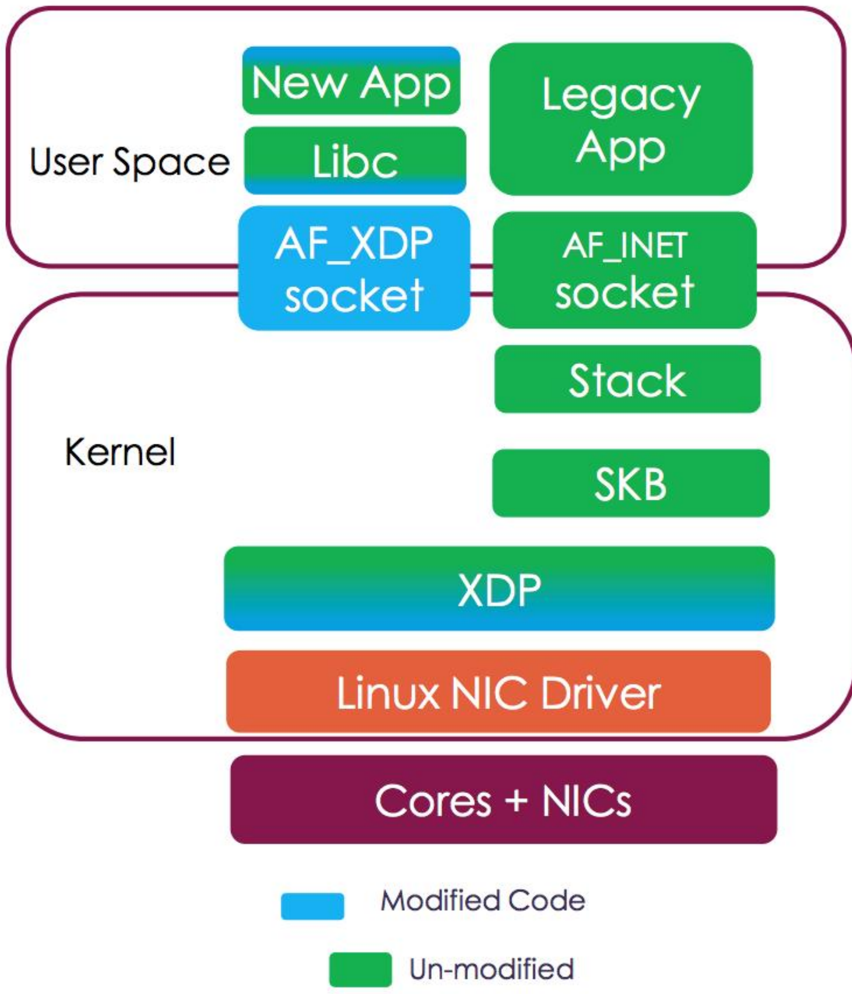


OVS Userspace Datapath (dpif-netdev)



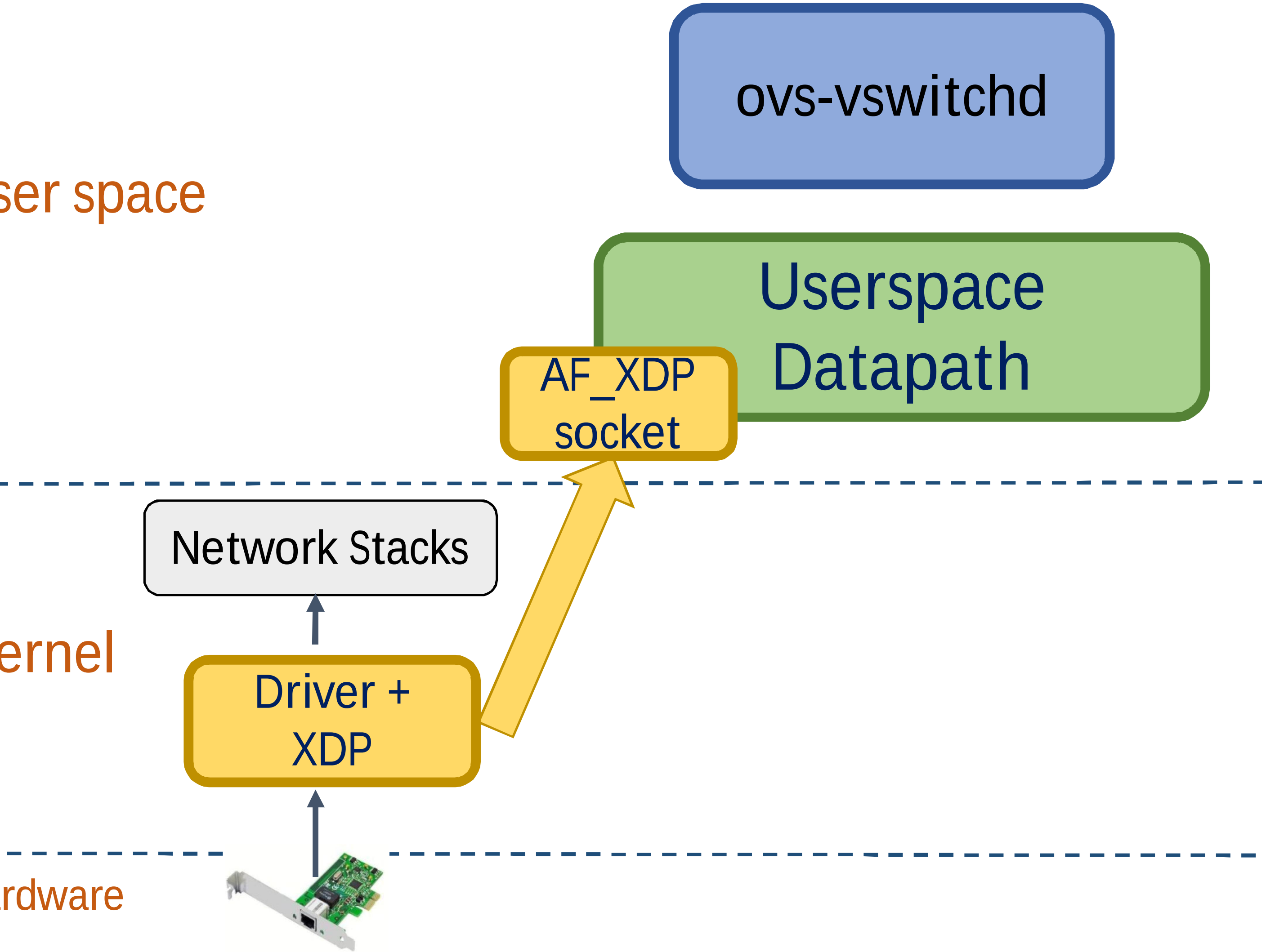
XDP and AF_XDP

- XDP: eXpress Data path
  - An eBPF hook point at the network device driver level
- AF_XDP:
  - A new socket type that receives/sends raw frames with high speed
  - Use XDP program to trigger receive
  - Userspace program manages Rx/Tx ring and Fill/Completion ring.
  - Zero Copy from DMA buffer to user space memory, umem



From “DPDK PMD for AF_XDP”

OVS-AF_XDP Project



Goal

- Use AF_XDP socket as a fa
channel to usersapce OVS
datapath
- Flow processing happens
userspace

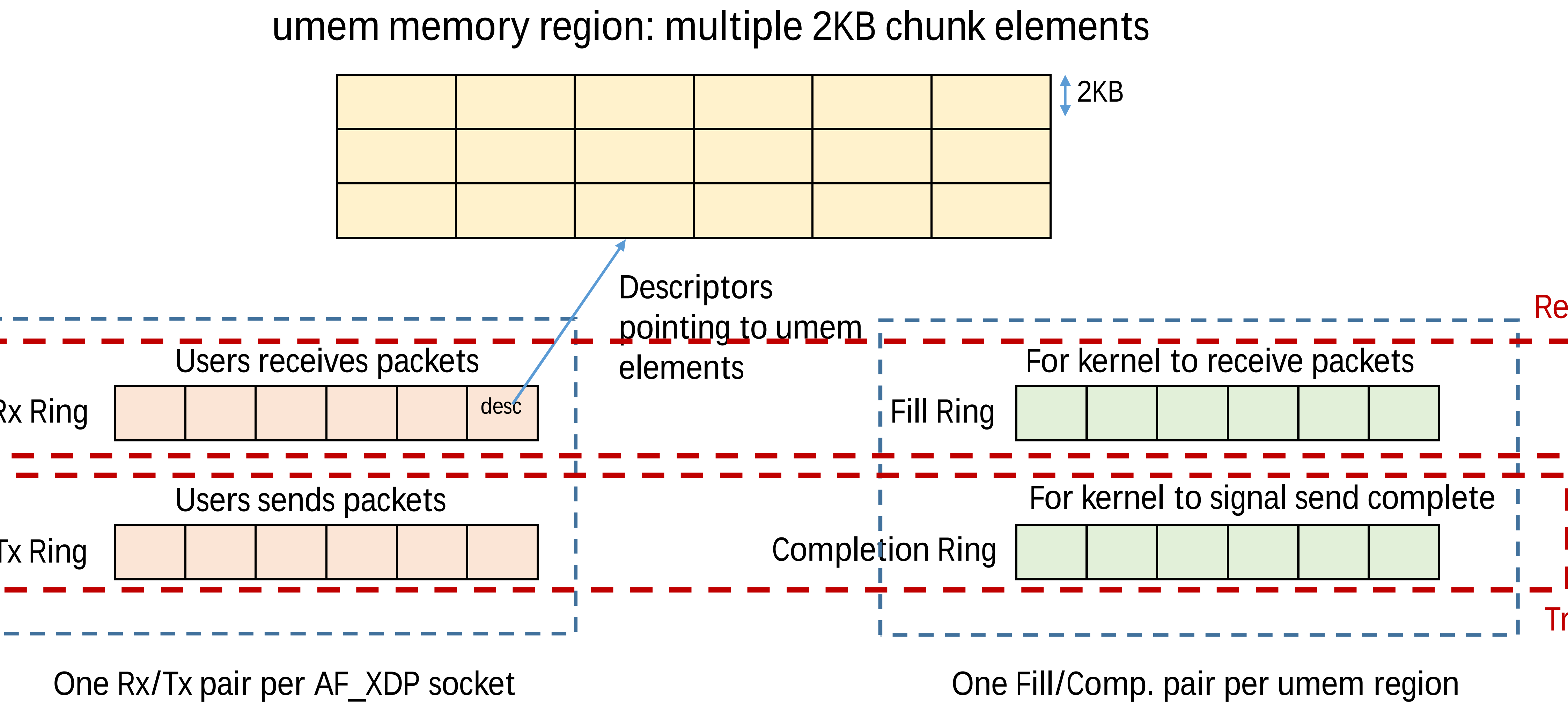
OVS-AF_XDP Architecture

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

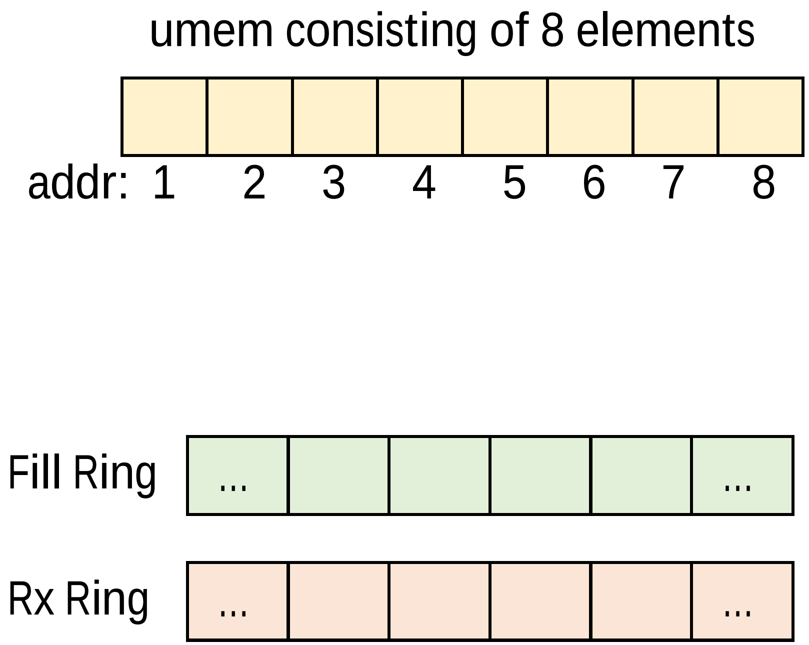
- Kernel: XDP program and eBPF map
- netdev: abstraction layer for network device
- AF_XDP netdev: implemtation of afxdp devi
- dpif: datapath interface
- dpif-netdev: userspace implementation of O datapath

Commands:
ovs-vsctl add-br br0
ovs-vsctl add-port br0 eth0 --
set int eth0 type="afxdp"

AF_XDP umem and rings Introduction

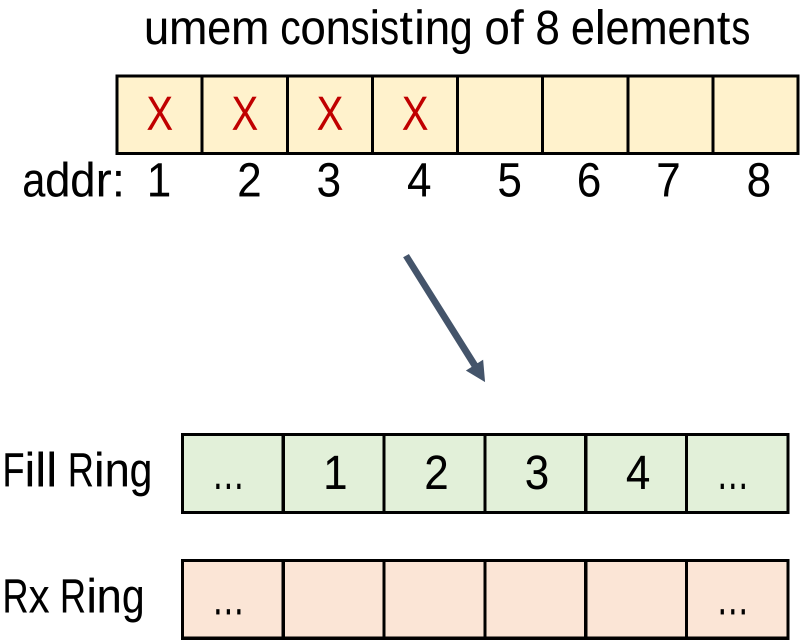


OVS-AF_XDP: Packet Reception (0)



Umem mempool =
{1, 2, 3, 4, 5, 6, 7, 8}

OVS-AF_XDP: Packet Reception (1)

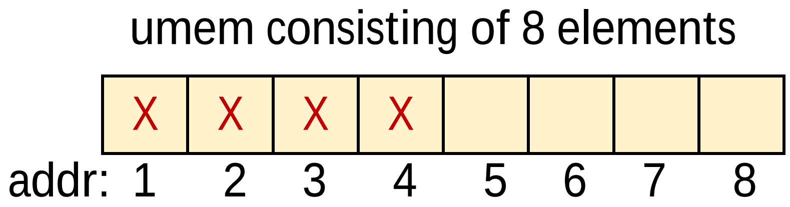


Umem mempool =
{5, 6, 7, 8}

X: elem in use

GET four elements,
program to Fill ring

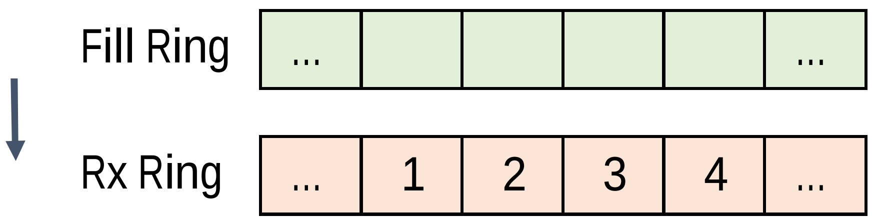
OVS-AFXDP: Packet Reception (2)



Umem mempool =
{5, 6, 7, 8}

X: elem in use

ernel receives four packets
ut them into the four umem chunks
ansition to Rx ring for users



OVS-AFXDP: Packet Reception (3)

umem consisting of 8 elements

	X	X	X	X	X	X	X	X
addr:	1	2	3	4	5	6	7	8

Umem mempool =
{}

X: elem in use

GET four elements
Program Fill ring

Fill Ring	...	5	6	7	8	...
Rx Ring	...	1	2	3	4	...

(so kernel can keeps receiving packets)

OVS-AFXDP: Packet Reception (4)

umem consisting of 8 elements

	X	X	X	X	X	X	X	X
addr:	1	2	3	4	5	6	7	8

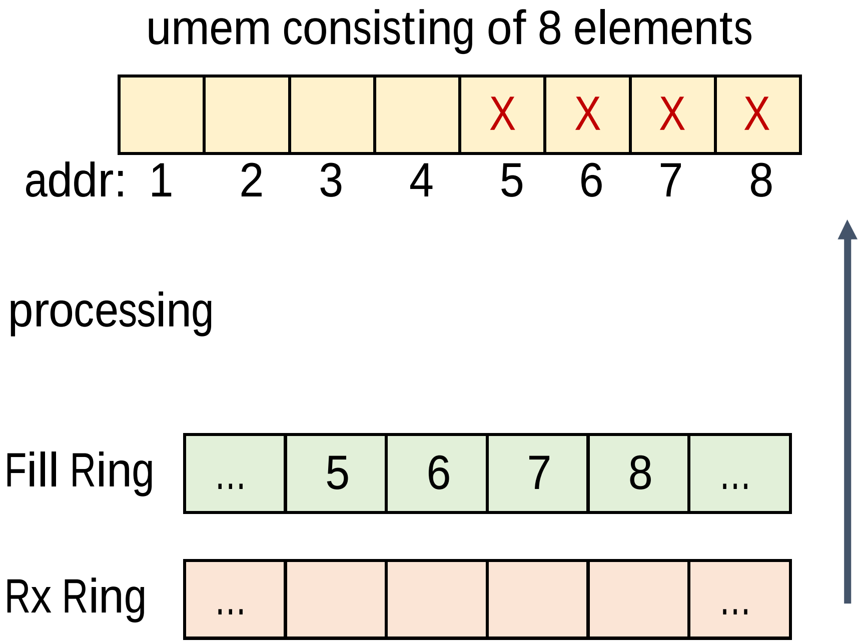
Umem mempool =
{}

X: elem in use

OVS userspace processes packets
on Rx ring

Fill Ring	...	5	6	7	8	...
Rx Ring	...	1	2	3	4	...

OVS-AFXDP: Packet Reception (5)

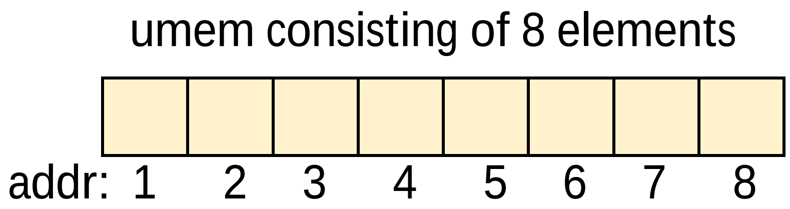


Umem mempool =
{1, 2, 3, 4}

X: elem in use

OVS userspace finishes packet processing
and recycle to umempool
Back to state (1)

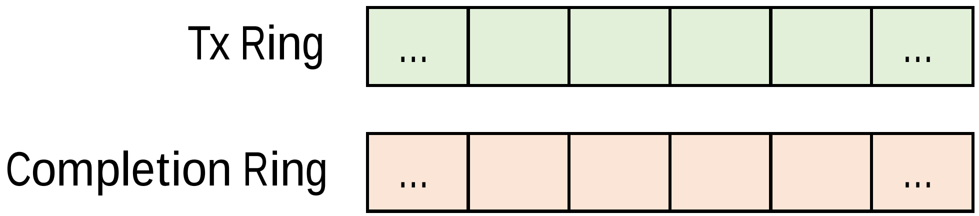
OVS-AFXDP: Packet Transmission (0)



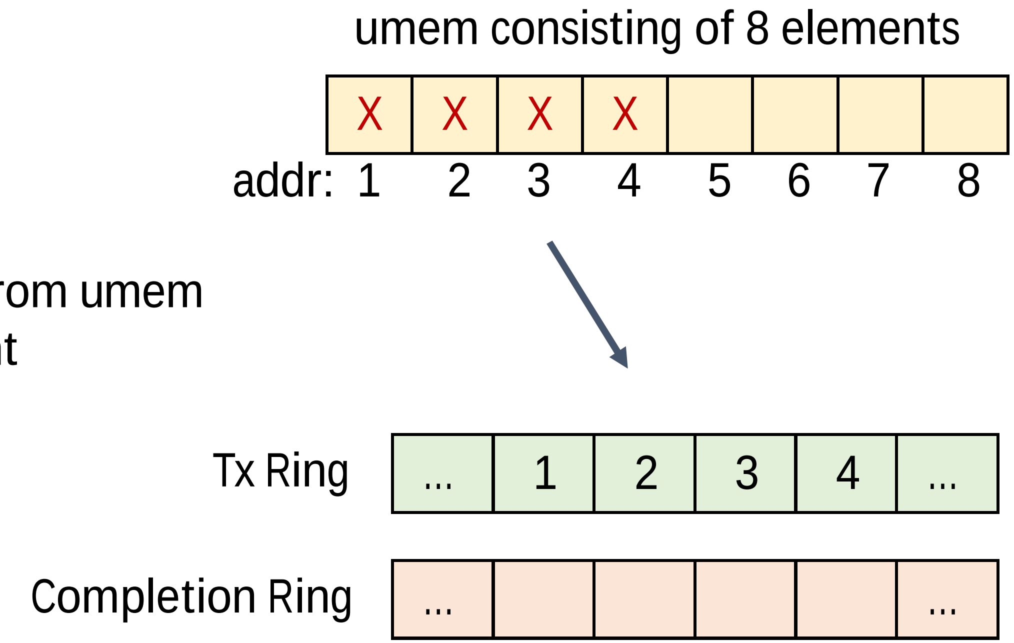
Umem mempool =
{1, 2, 3, 4, 5, 6, 7, 8}

X: elem in use

OVS userspace has four packets to send



OVS-AFXDP: Packet Transmission (1)



Umem mempool =
{5, 6, 7, 8}

X: elem in use

GET fours element from umem
Copy packets content
Place in Tx ring

OVS-AFXDP: Packet Transmission (2)

umem consisting of 8 elements

	X	X	X	X				
addr:	1	2	3	4	5	6	7	8

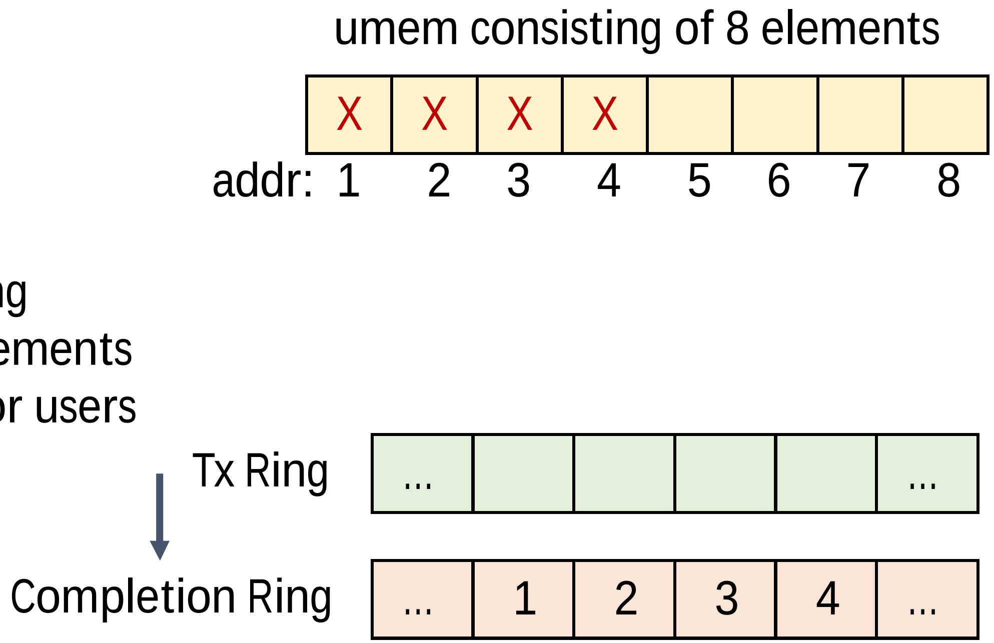
Umem mempool =
{5, 6, 7, 8}

X: elem in use

Issue sendmsg() syscall
Kernel tries to send packets on Tx ring

Tx Ring	...	1	2	3	4	...
Completion Ring	...					...

OVS-AFXDP: Packet Transmission (3)

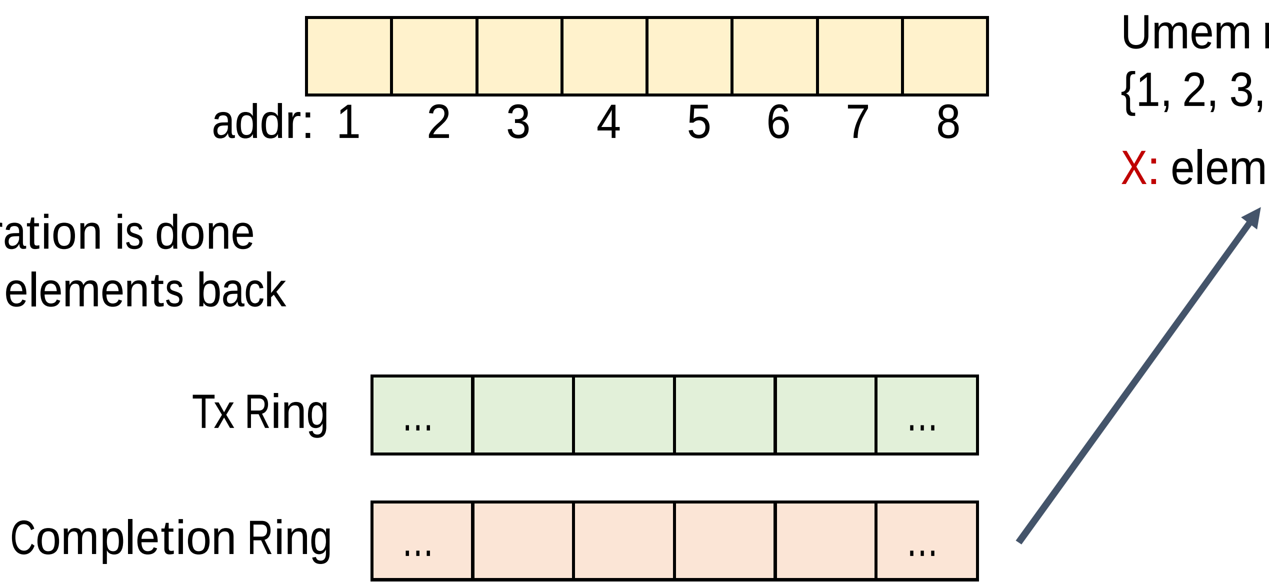


Umem mempool = {5, 6, 7, 8}

X: elem in use

Kernel finishes sending
Transition the four elements to Completion Ring for users

OVS-AFXDP: Packet Transmission (4)



OVS knows send operation is done
Recycle/PUT the four elements back
to umempool

Umem mempool =
{1, 2, 3, 4, 5, 6, 7, 8}

X: elem in use

Optimizations

- OVS pmd (Poll-Mode Driver) netdev for rx/tx
  - Before: call poll() syscall and wait for new I/O
  - After: dedicated thread to busy polling the Rx ring
- UMEM memory pool
  - Fast data structure to GET and PUT umem elements
- Packet metadata allocation
  - Before: allocate md when receives packets
  - After: pre-allocate md and initialize it
- Batching sendmsg system call

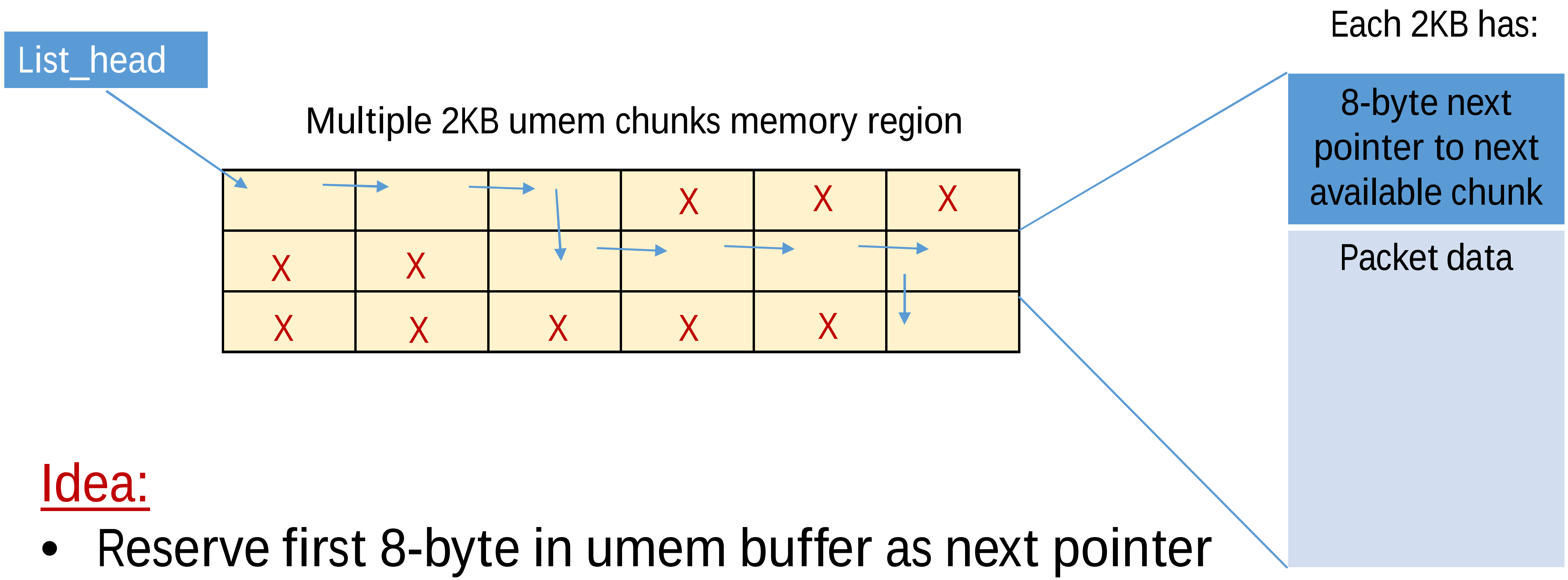
Umempool Design

- umempool keeps track of available umem elements
  - GET: take out N umem elements
  - PUT: put back N umem elements
- Every ring access need to call umem element GET/PUT

Three designs:

- LILO-List_head: embed in umem buffer, linked by a list_head, push/pop style
- FIFO-ptr_ring: a pointer ring with head and tail pointer
- LIFO-ptr_array: a pointer array and push/pop style access

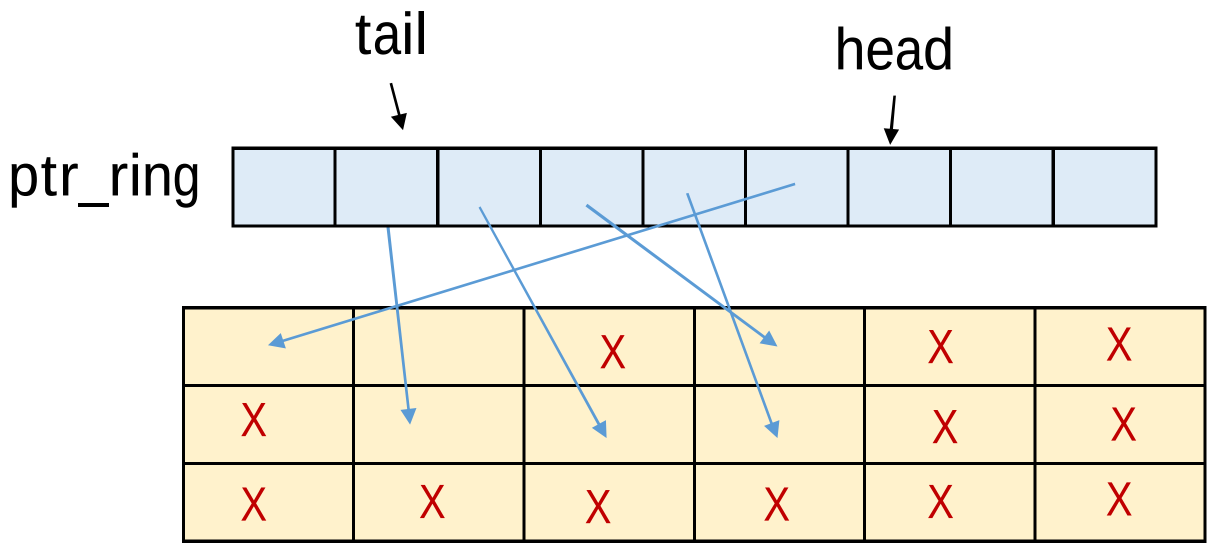
LILO-list_head Design



Idea:

- Reserve first 8-byte in umem buffer as next pointer
- No extra mempool metadata allocation needed
- GET from the head and PUT from the head

FIFO-ptr_ring Design

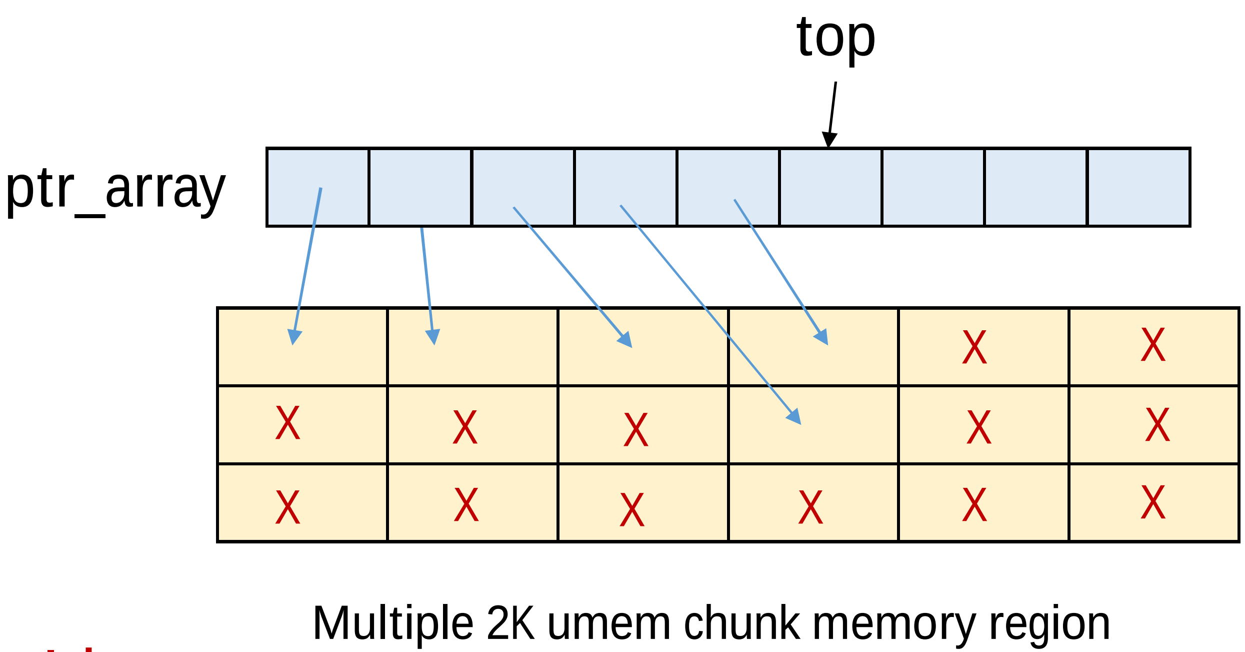


Multiple 2KB umem chunk memory region

Idea:

- Allocate a ring with each ring element contains umem address
- Producer: PUT elements at ptr_ring[head] and head++
- Consumer: GET elements and ptr_ring[tail] and tail++

LIFO-ptr_array Design



Idea:

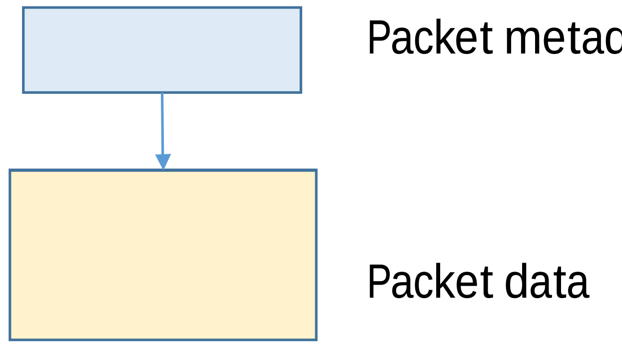
- Each ptr_array element contains a umem address
- Producer: PUT elements on top and top++
- Consumer: GET elements from top and top--

Packet Metadata Allocation

- Every packets in OVS needs metadata: struct dp_packet
- Initialize the packet data independent fields

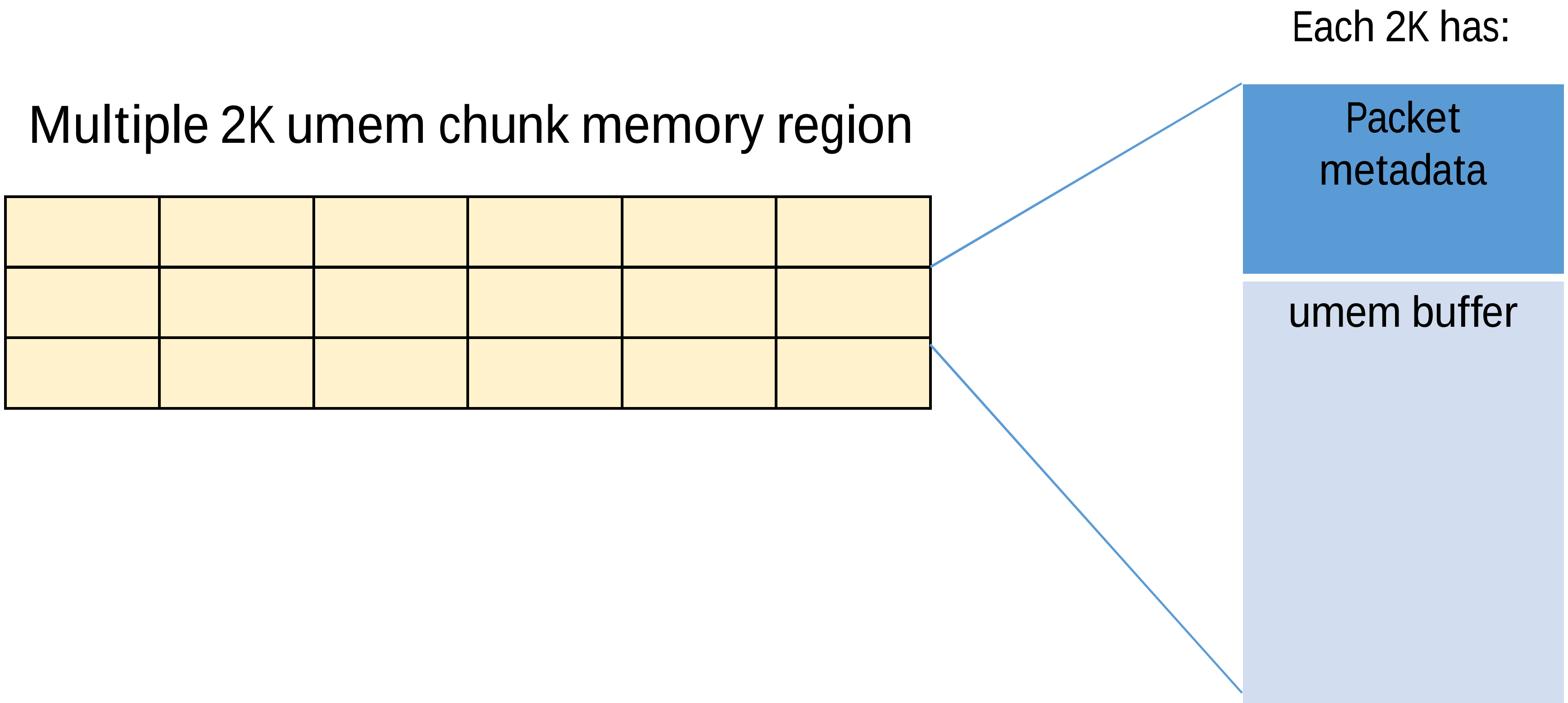
<u>Two designs:</u>

1. Embedding in umem packet buffer:
   - Reserve first 256-byte for struct dp_packet
   - Similar to DPDK mbuf design
2. Separate from umem packet buffer:
   - Allocate an array of struct dp_packet
   - Similar to skb_array design



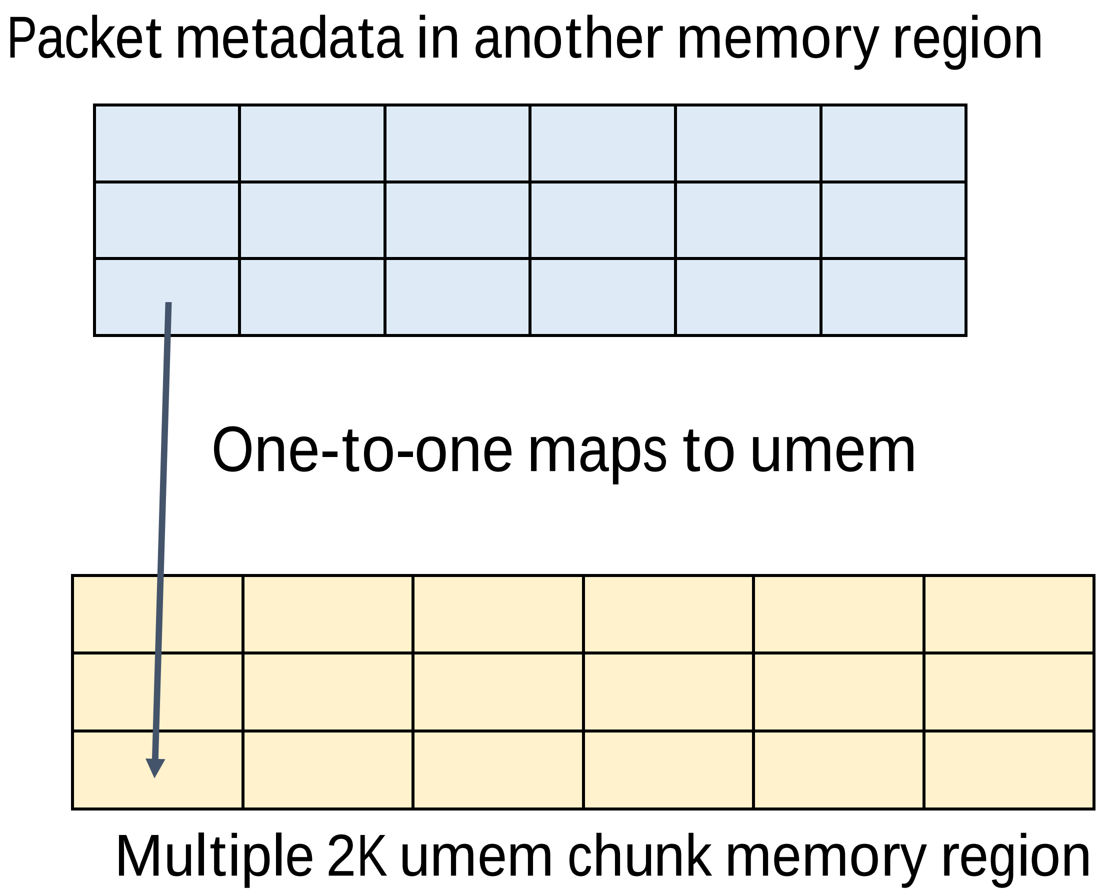
Packet Metadata Allocation

Embedding in umem packet buffer

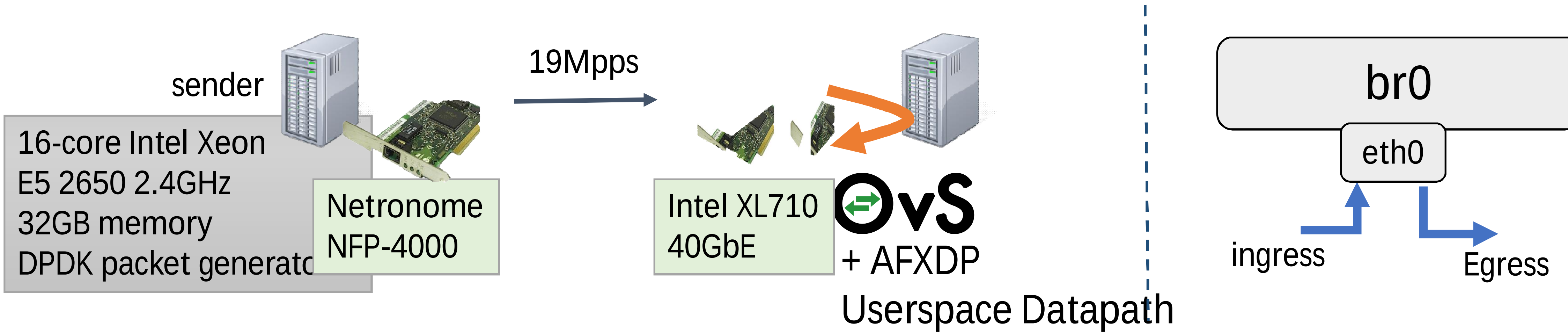


Packet Metadata Allocation

Separate from umem packet buffer



Performance Evaluation



- Sender sends 64Byte, 19Mpps to one port, measure the receiving packet rate at the other port
- Measure single flow, single core performance with Linux kernel 4.19-rc3 and OVS 2.9
- Enable AF_XDP Zero Copy mode

Performance Evaluation

Experiments

- OVS-AFXDP
  - rxdrop: parse, lookup, and action = drop
  - L2fwd: parse, lookup, and action = set_mac, output to the received port
- XDPSOCK: AF_XDP benchmark tool
  - rxdrop/l2fwd: simply drop/fwd without touching packets
- LIFO-ptr_array + separate md allocation shows the best

Results

	XDPSOCK	OVS-AFXDP
rxdrop	19Mpps	19Mpps
l2fwd	17Mpps	14Mpps

Conclusion and Discussion

Future Work

- Follow up new kernel AF_XDP's optimizations
- Try virtual devices vhost/virtio with VM-to-VM traffic
- Bring feature parity between userspace and kernel datapath

Discussion

- Usage model: # of XSK, # of queue, # of pmd/non-pmd
- Comparison with DPDK in terms of deployment difficulty

Dislike?

Like?

Question?

Thank You

Batching sendmsg syscall

- Place a batch of 32 packets on TX ring, issue send syscall
- Design 1
- Check this 32 packets on completion ring, then recycle
- If not, keep issuing send
- Design 2
- Check any 32 packets on completion ring, then recycle
- If not, keep issuing send
- Design 3
- Issue sendmsg syscall