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

Datapath

Fast Path

Slow Path
OVS Linux Kernel Datapath

- Slow path in userspace
  - socket
  - ovs-vswitchd
  - OVS Kernel module

- Fast Path in Kernel
  - IP/routing
  - Device RX Hook
  - driver

- Hardware
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 often non-obvious to fix

- Implement datapath functionalities in eBPF
  - More stable ABI and guarantee to run in newer kernel
  - More opportunities for experiments / innovations
What is eBPF?

• An in-kernel virtual machine
  • Users can load its program and attach to a specific hook point in kernel
  • Safety guaranteed by BPF verifier
  • Attach points: network, trace point, driver, ... etc

• Maps
  • Efficient key/value store resides in kernel space
  • Can be shared between eBPF program 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**

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

**Diagram**
- Slow path in userspace
- Fast Path in Kernel
- IP/routing
- TC hook
- Parse
- Lookup
- Actions
- driver
- Hardware
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 (max: 512-byte, sw_flow_key: 464-byte)
• 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

**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 to trigger lookup again

**Fast Path**
- Subsequent packets hit flow in eBPF map

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

1. Ingress
2. miss upcall (perf ring buf -> netlink)
3. flow installation (netlink TLV -> fixed array -> eBPF map)
4. actions

Parser
Flow Table (eBPF hash map)

ovs-vswitchd
Review: OVS Kernel Datapath Actions

A list of actions to execute on the packet

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 **single flow, single core** performance with Linux kernel 4.9-rc3 on OVS server
# OVS Kernel and eBPF Datapath Performance

<table>
<thead>
<tr>
<th>OVS Kernel DP Actions</th>
<th>Mpps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>1.34</td>
</tr>
<tr>
<td>Set dst_mac + Output</td>
<td>1.23</td>
</tr>
<tr>
<td>Set GRE tunnel + Output</td>
<td>0.57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>eBPF DP Actions</th>
<th>Mpps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redirect (no parser, lookup, actions)</td>
<td>1.90</td>
</tr>
<tr>
<td>Output</td>
<td>1.12</td>
</tr>
<tr>
<td>Set dst_mac + Output</td>
<td>1.14</td>
</tr>
<tr>
<td>Set GRE tunnel + Output</td>
<td>0.48</td>
</tr>
</tbody>
</table>

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
• Taking existing features and converting to eBPF is hard
• OVS datapath logic is difficult
OVS-AF_XDP
OVS-AF_XDP Motivation

• Pushing all OVS datapath features into eBPF is not easy
  • A large flow key on stack
  • Variety of protocols and actions
  • Dynamic number of actions applied for each flow

• **Ideas**
  • Retrieve packets from kernel as fast as possible
  • Do the rest of the processing in userspace

• **Difficulties**
  1. Reimplement all features in userspace
  2. Performance
OVS Userspace Datapath (dpif-netdev)

SDN Controller

ovs-vswitchd

Userspace Datapath

Both slow and fast path in userspace

Another datapath implementation in userspace

DPDK library

Hardware
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, achieving line rate (14Mpps)!

From “DPDK PMD for AF_XDP”
OVS-AF_XDP Project

Goal

- Use AF_XDP socket as a fast channel to userspace OVS datapath
- Flow processing happens in userspace
AF_XDP umem and rings Introduction

umem memory region: multiple 2KB chunk elements

- Users receives packets
- Users sends packets

Descriptors pointing to umem elements

- Rx Ring
- Tx Ring

One Rx/Tx pair per AF_XDP socket

For kernel to receive packets

- Fill Ring
- Completion Ring

One Fill/Comp. pair per umem region
AF_XDP umem and rings Introduction

umem memory region: multiple 2KB chunk elements

Descriptors pointing to umem elements

Receive

Users receives packets
Rx Ring

For kernel to receive packets
Fill Ring

Transmit

Users sends packets
Tx Ring

For kernel to signal send complete
Completion Ring

One Rx/Tx pair per AF_XDP socket
One Fill/Comp. pair per umem region
OVS-AF_XDP: Packet Reception (0)

umem consisting of 8 elements

addr: 1 2 3 4 5 6 7 8

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

Fill Ring

Rx Ring
OVS-AF_XDP: Packet Reception (1)

- umem consisting of 8 elements
  - addr: 1 2 3 4 5 6 7 8
  - Umem mempool = \{5, 6, 7, 8\}
  - X: elem in use

- GET four elements, program to Fill ring

- Fill Ring
  - ... 1 2 3 4 ...

- Rx Ring
  - ... ... ... ...
Kernel receives four packets
Put them into the four umem chunks
Transition to Rx ring for users

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

X: elem in use

Kernel receives four packets
Put them into the four umem chunks
Transition to Rx ring for users

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

X: elem in use
OVS-AFXDP: Packet Reception (3)

umem consisting of 8 elements

addr: 1 2 3 4 5 6 7 8

GET four elements
Program Fill ring

Fill Ring

Rx Ring

X: elem in use

Umem mempool = {}

X: elem in use

(so kernel can keeps receiving packets)
OVS-AFXDP: Packet Reception (4)

OVS userspace processes packets on Rx ring

Umem mempool = {}
X: elem in use

umem consisting of 8 elements

<table>
<thead>
<tr>
<th>addr:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Fill Ring

| ... | 5 | 6 | 7 | 8 | ... |

Rx Ring

| ... | 1 | 2 | 3 | 4 | ... |
OVS-AFXDP: Packet Reception (5)

umem consisting of 8 elements

addr: 1 2 3 4 5 6 7 8

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

Fill Ring

... 5 6 7 8 ...

Rx Ring

... ...

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

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: A freelist keeps tracks of free buffers
  - 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 (BEST!)
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 packet in OVS needs metadata: struct dp_packet
• Initialize the packet data independent fields

Two designs:
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
Separate from umem packet buffer

Packet metadata in another memory region

One-to-one maps to umem

Multiple 2K umem chunk memory region
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

• XDP_sock: AF_XDP benchmark tool
  • rxdrop/l2fwd: simply drop/fwd without touching packets

• LIFO-ptr_array + separate md allocation shows the best

Results

<table>
<thead>
<tr>
<th></th>
<th>XDP_sock</th>
<th>OVS-AFXDP</th>
<th>Linux Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>rxdrop</td>
<td>19Mpps</td>
<td>19Mpps</td>
<td>&lt; 2Mpps</td>
</tr>
<tr>
<td>l2fwd</td>
<td>17Mpps</td>
<td>14Mpps</td>
<td>&lt; 2Mpps</td>
</tr>
</tbody>
</table>
Conclusion and Discussion

Future Work
• Try virtual devices vhost/virtio with VM-to-VM traffic
• Bring feature parity between userspace and kernel datapath

Discussion
• Balance CPU utilization of pmd/non-pmd
• Comparison with DPDK in terms of deployment difficulty
# Comparison

<table>
<thead>
<tr>
<th></th>
<th>OVS-eBPF</th>
<th>OVS-AF_XDP</th>
<th>OVS Kernel Module</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maintenance Cost</strong></td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Comparable with kernel</td>
<td>High with cost of CPU</td>
<td>Standard (&lt; 2Mpps)</td>
</tr>
<tr>
<td><strong>Development Efforts</strong></td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td><strong>New feature deployment</strong></td>
<td>Easy</td>
<td>Easy</td>
<td><strong>Hard</strong> due to ABI change</td>
</tr>
<tr>
<td><strong>Safety</strong></td>
<td>High due to verifier</td>
<td>Depends on reviewers</td>
<td>Depends on reviewers</td>
</tr>
</tbody>
</table>
Thank You

Question?