The Design and Implementation of Open vSwitch

Ben Pfaff*
Justin Pettit*
Teemu Koponen*
Ethan J. Jackson*
Andy Zhou*
Jarno Rajahalme*
Jesse Gross*
Alex Wang*
Jonathan Stringer*
Pravin Shelar*
Keith Amidon†
Martin Casado*

*VMware
†Awake Networks
What is Open vSwitch?

From openvswitch.org:

“Open vSwitch is a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols (e.g. NetFlow, sFlow, SPAN, RSPAN, CLI, LACP, 802.1ag).”
Where is Open vSwitch Used?

• Broad support:
  – Linux, FreeBSD, NetBSD, Windows, ESX
  – KVM, Xen, Docker, VirtualBox, Hyper-V, …
  – OpenStack, CloudStack, OpenNebula, …

• Widely used:
  – Most popular OpenStack networking backend
  – Default network stack in XenServer
  – 1,440 hits in Google Scholar
  – Thousands of subscribers to OVS mailing lists
Open vSwitch Architecture

- **VMs**
  - VM 1
  - VM 2
  - VM n

- **Hypervisor**
  - ovsdb-server
  - ovs-vswitchd
  - kernel module

- **Controller**

- **NICs**

- **OVSDB**
- **OpenFlow**
- **Netlink**
- **User**
- **Kernel**
Use Case: Network Virtualization

OpenFlow tables

<table>
<thead>
<tr>
<th>Table 0</th>
<th>Table 1</th>
<th>Table 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow 1</td>
<td>Flow 1</td>
<td>Flow 1</td>
</tr>
<tr>
<td>Flow 2</td>
<td>Flow 2</td>
<td>Flow 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Packet ingress

Physical to Logical

L2 Lookup

Logical to Physical

OpenFlow Pipeline

packet egress
Implications for Forwarding Performance

OpenFlow tables

<table>
<thead>
<tr>
<th>Table 0</th>
<th>Table 1</th>
<th>Table 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow 1</td>
<td>Flow 1</td>
<td>Flow 1</td>
</tr>
<tr>
<td>Flow 2</td>
<td>Flow 2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Physical to Logical

L2 Lookup

Logical to Physical

k₀ hash lookups

k₁ hash lookups

k₂₄ hash lookups

100+ hash lookups per packet for tuple space search?
Non-solutions

• All of these helped:
  – Multithreading
  – Userspace RCU
  – Batching packet processing
  – Classifier optimizations
  – Microoptimizations

• None of it helped enough: % versus x.

Classification is expensive on general-purpose CPUs!
OVS Cache v1: Microflow Cache

Microflow:
- Complete set of packet headers and metadata
- Suitable for hash table
- Shaded data below:
  
<table>
<thead>
<tr>
<th>Eth</th>
<th>IP</th>
<th>TCP</th>
<th>payload</th>
</tr>
</thead>
</table>

OpenFlow Tables

hit

miss

Microflow Cache

OpenFlow Controller (in theory)
Speedup with Microflow Cache

From 100+ hash lookups per packet, to just 1!
Microflow Caching in Practice

• Tremendous speedup for most workloads
• Problematic traffic patterns:
  – Port scans
    • Malicious
    • Accidental (!)
  – Peer-to-peer rendezvous applications
  – Some kinds of network testing
• All of this traffic has lots of short-lived microflows
  – Fundamental caching problem: low hit rate
Using a More Expensive Cache

If $k_c \ll k_0 + k_1 + \ldots + k_{24}$: benefit!
Naive Approach to Populating Cache

Combine tables 0...24 into one flow table. Easy! Usually, $k_c << k_0 + k_1 + \ldots + k_{24}$. But:

Table 0
- ip_src=a
- ip_src=b
- ip_src=c
- ip_src=d
- $n_1$ flows

Table 1
- ip_dst=e
- ip_dst=f
- ip_dst=g
- ip_dst=h
- $n_2$ flows

Table 24
- eth_dst=i
- eth_dst=j
- eth_dst=k
- eth_dst=m
- $n_{24}$ flows

Table 0+1+...+24
- ip_src=a, ip_dst=e, ..., eth_dst=i
- ip_src=a, ip_dst=e, ..., eth_dst=j
- ip_src=a, ip_dst=e, ..., eth_dst=k
- ... 
- ip_src=d, ip_dst=h, ..., eth_dst=k
- ip_src=d, ip_dst=h, ..., eth_dst=m
- $n_{1} \times n_{2} \times \ldots \times n_{24}$ flows

“Crossproduct Problem”
Lazy Approach to Populating Cache

Solution: Build cache of combined “megaflows” lazily as packets arrive.

Table 0
- $ip_{src}=a$
- $ip_{src}=b$
- $ip_{src}=c$
- $ip_{src}=d$

$n_1$ flows

Table 1
- $ip_{dst}=e$
- $ip_{dst}=f$
- $ip_{dst}=g$
- $ip_{dst}=h$

$n_2$ flows

Table 24
- $eth_{dst}=i$
- $eth_{dst}=j$
- $eth_{dst}=k$
- $eth_{dst}=m$

$n_{24}$ flows

Megaflow Cache
- $ip_{src}=a, ip_{dst}=f, ..., eth_{dst}=i$
- $ip_{src}=c, ip_{dst}=g, ..., eth_{dst}=k$
- $ip_{src}=d, ip_{dst}=e, ..., eth_{dst}=m$

populated dynamically only from actually observed packets

Same (or better!) table lookups as naive approach.
Traffic locality yields practical cache size.
OVS Cache v2: “Megaflow” Cache
Making Megaflows Better

• Megaflows are more effective when they match fewer fields.
  – Megaflows that match TCP ports are almost like microflows!
  – Described approach matches every field that appears in any flow table

• Requirements:
  – online
  – fast

• Contribution: Megaflow generation improvements (Section 5).
Megaflow vs. Microflow Cache Performance

- Microflow cache:
  - $k_0 + k_1 + \cdots + k_{24}$ lookups for first packet in microflow
  - 1 lookup for later packets in microflow

- Megaflow cache:
  - $k_c$ lookups for (almost) every packet

- $k_c > 1$ is normal, so megaflows perform worse in common case!

- Best of both worlds would be:
  - $k_c$ lookups for first packet in microflow
  - 1 lookup for later packets in microflow
Parting Thoughts

- Architectural tension: expressibility vs. performance
- OpenFlow is expressive but troublesome to make fast on x86
  - Performance requirements can make applications avoid OpenFlow
- Caching provides OVS with expressibility and performance
- Applications can freely evolve decoupled from performance
  - Specialized code would be slower!
- Starting from a more general problem produced better results