The Design and Implementation of Open vSwitch

Antoni Rościszewski

November 7, 2018

Table of Contents

Research paper

The Design and Implementation of Open vSwitch
Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, and Pravin Shelar, Keith Amidon, Martín Casado

https://www.usenix.org/node/188961

What is Open vSwitch?

- Open vSwitch (OVS) is a production quality virtual switch platform that supports standard management interfaces and opens the forwarding functions to programmatic extension and control.
- supports multiple Linux-based virtualization technologies including Xen/XenServer, KVM, and VirtualBox
- open source: https://github.com/openvswitch/ovs

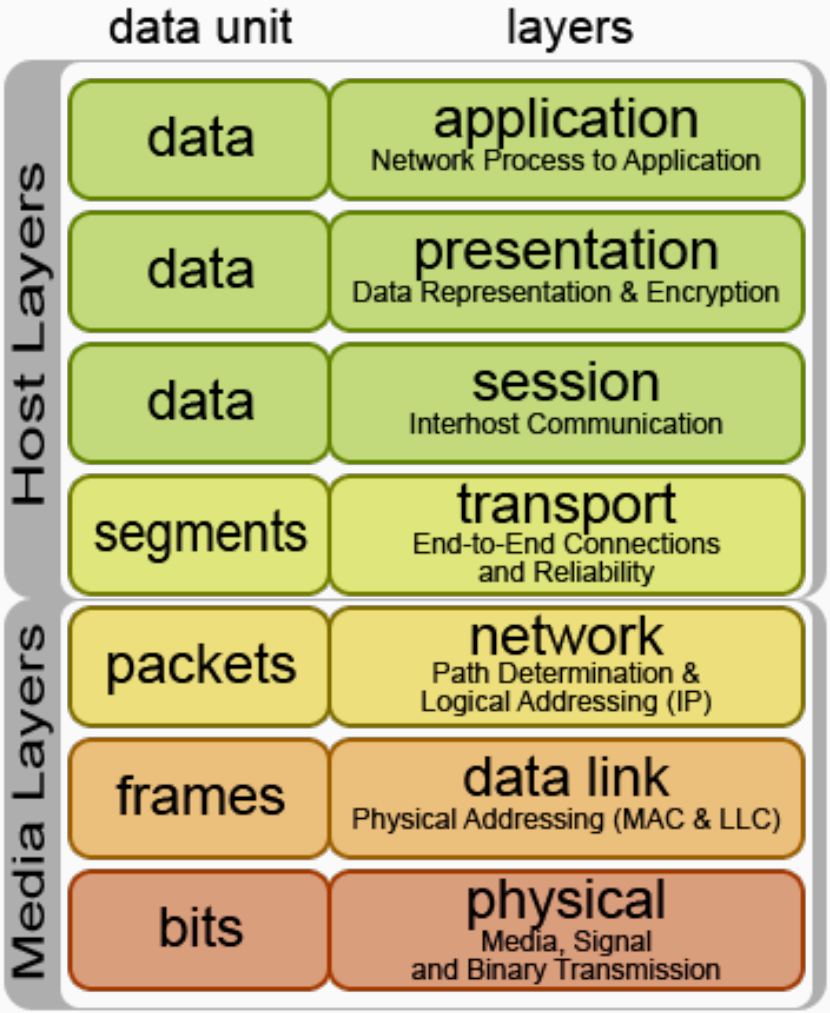
Introduction

Introduction

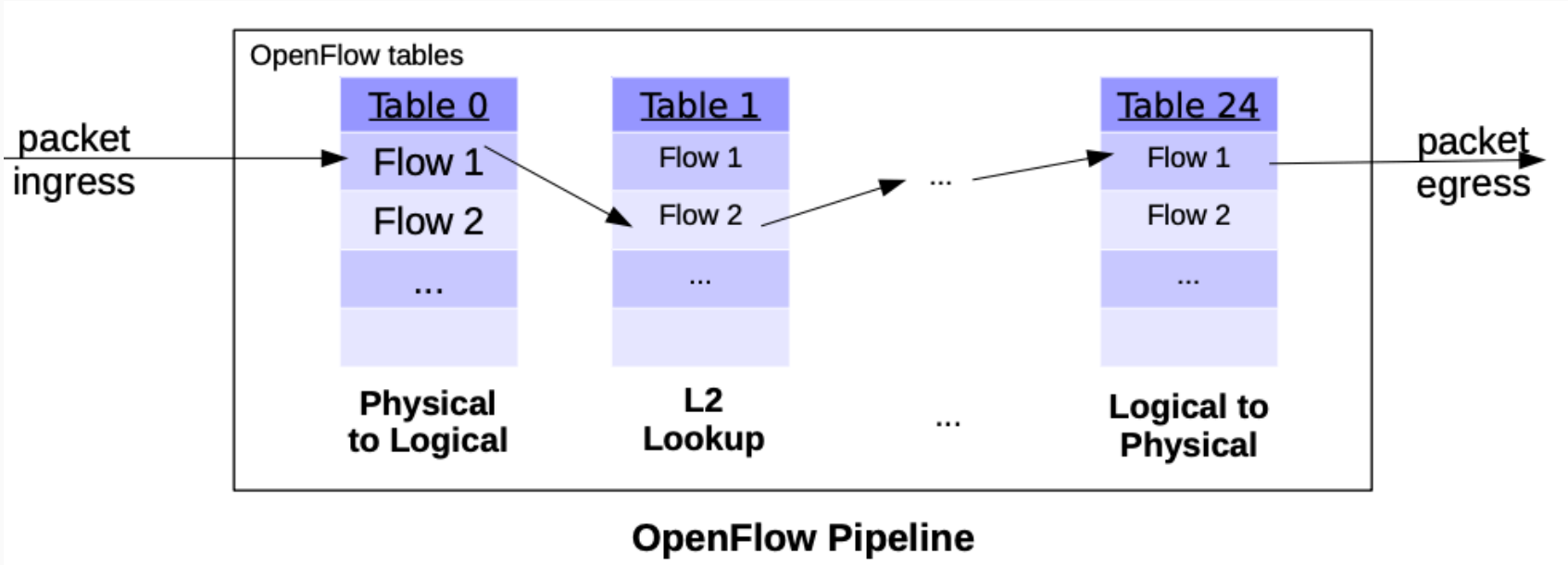
Abbreviations

- **OpenFlow** - communications protocol that gives access to the forwarding plane of a network switch or router over the network
- **NVP** - Nicira Network Virtualization Platform
- **OVS** - Open vSwitch

OSI model



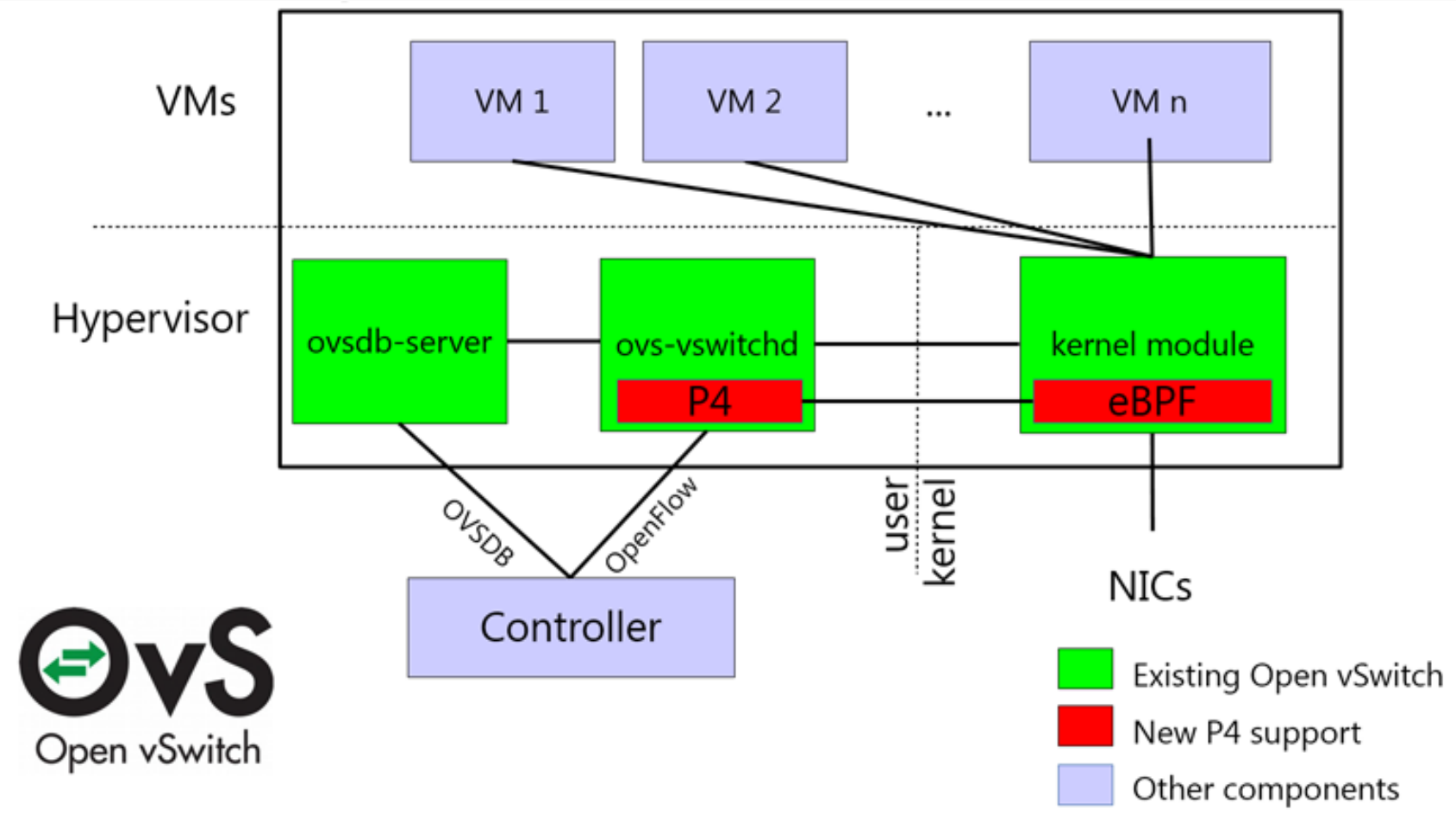
Network virtualization



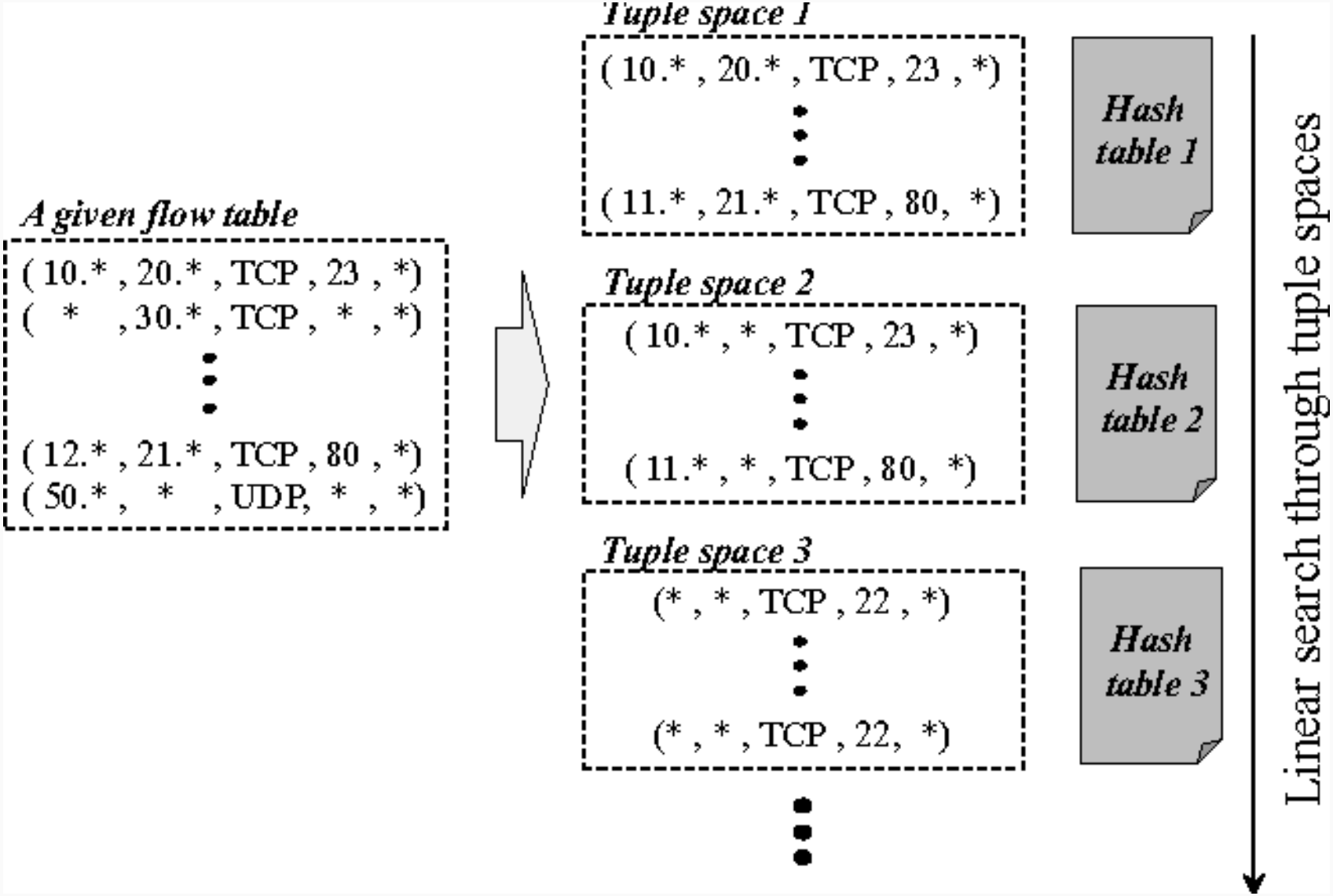
Main issues

- in the forwarding context, it requires many hash lookups
- 100% performance increase is not enough, 100x increase was needed

Open vSwitch architecture



Tuple space search



Flow Cache design

Microflow cache

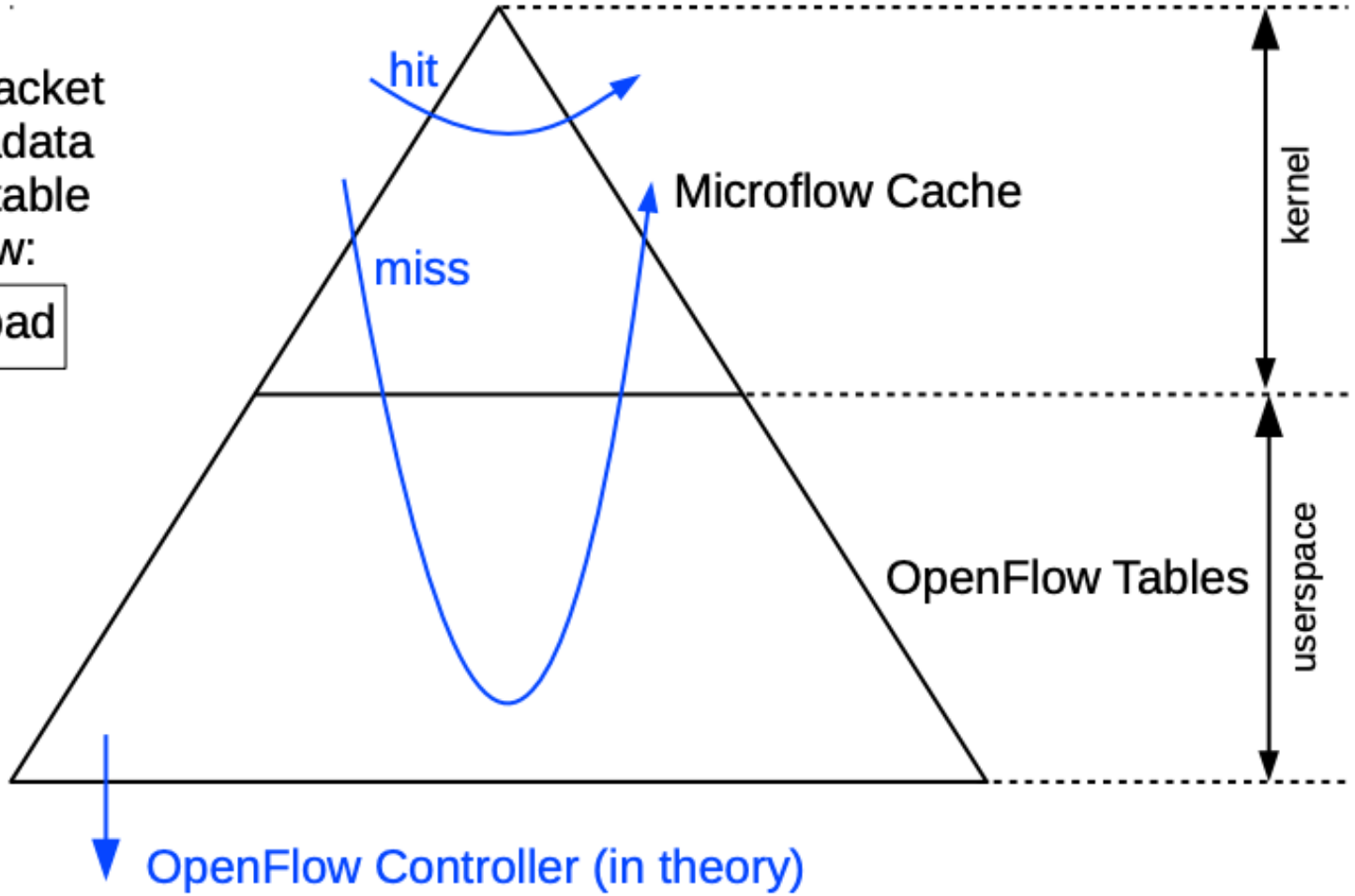
- started as a Linux kernel module, which:
  1. received a packet from a NIC or VM
  2. classified through the OpenFlow table
  3. modified it as necessary
  4. sent it to another port
- later reimplemented as *microflow cache*: single cache entry exact matches with all the packet header fields supported by OpenFlow (internally a simple hash table instead of a complicated classifier)

Microflow cache

Microflow:

- Complete set of packet headers and metadata
- Suitable for hash table
- Shaded data below:

Eth	IP	TCP	payload



Open vSwitch

Megaflow cache (v2)

- *microflow caching* suffered serious performance degradation for large numbers of short lived connections ($\rightarrow$ cache misses, unneeded packet classification, userspace-kernel communication)
- *megaflow cache*: single flow lookup table, which supports generic matching, i.e., it caching forwarding decisions for larger aggregates of traffic than connections (i.e. arbitrary packet field matching)
- simpler and lighter runtime:
  1. no priorities
  2. one megaflow classifier (instead of a pipeline)

Megaflow cache: naive approach

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

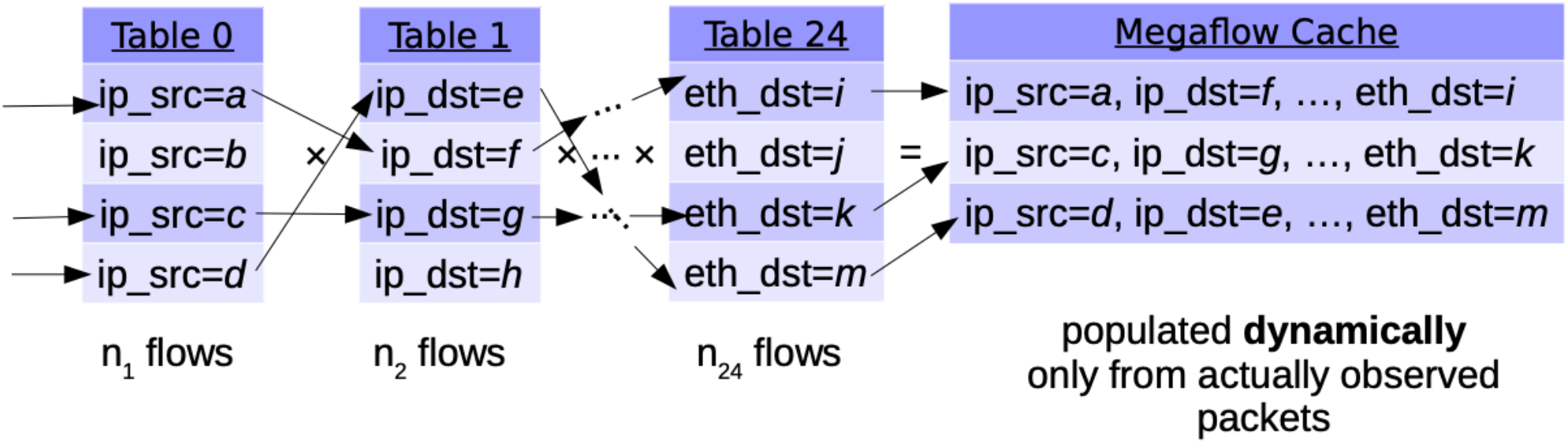
=

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*

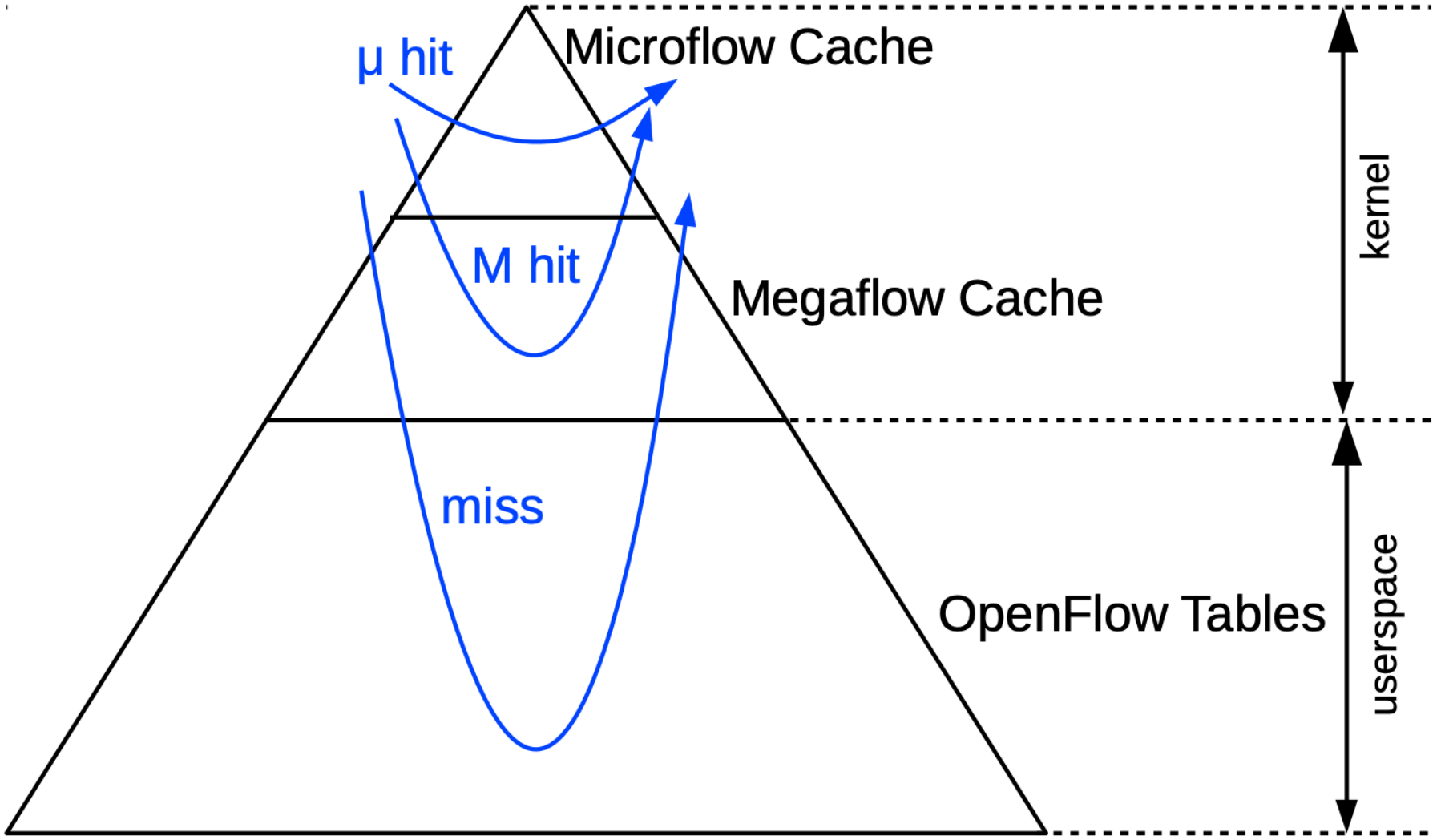
up to $n_1 \times n_2 \times \cdots \times n_{24}$ flows

Megaflow cache: lazy/dynamic approach

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



Dual caches (final design)



Tuple priority sorting (1)

- lookup in a tuple space search classifier ordinarily requires searching every tuple
- solution: track (in each tuple T) the maximum priority T.pri_max of any flow entry in T

```
function PrioritySortedTupleSeach(H)
    B ← NULL /* Best flow match so far. */
    for tuple T in descending order of T.pri max do
        if B != NULL and B.pri ≥ T.pri max then
            return B
    if T contains a flow F matching H then
        if B = NULL or F.pri > B.pri then
            B ← F
    return B
```

Tuple priority sorting (2)

- OpenFlow table installed by a production deployment of VMware's NVP controller
- 29 tuples, 26 of which contained flows of a single priority
- when searching in descending priority order, one can always terminate immediately following a successful match in such a tuple
- considering the other tuples, two contained flows with two unique priorities that were higher than those in any subsequent tuple, so any match in either of these tuples terminated the search
- final tuple: flows with five unique priorities ranging from 32767 to 36866; in the worst case, if the lowest priority flows matched in this tuple, then the remaining tuples with T.pri max > 32767 (up to 20 tuples based on this tuple's location in the sorted list), must also be searched.

Staged lookup

- Problem: tuple space search searches each tuple with a hash table lookup $\rightarrow$ megaflow must match all the bits of fields included in the tuple, even if the tuple search fails
- Other data structures considered: A trie would allow a search on any prefix of fields, but it would also increase the number of memory accesses required by a successful search from $\mathcal{O}(1)$ to $\mathcal{O}(n)$ in the length of the tuple fields
- data structures larger than $\mathcal{O}(n)$ (n - # of flows in a tuple), as OpenFlow tables can have hundreds of thousands of flows
- Solution: changed each tuple from a single hash table to an array of four hash tables, called stages: metadata, L2, L3, and L4
- Lookup in a tuple searches each of its stages in order. If any search turns up no match, then the overall search of the tuple also fails, and only the fields included in the stage last searched must be added to match

Prefix Tracking

- IPv4 and IPv6 subnet matching problem (during routing):
  - when all the flows that match on such a field use the **same subnet size**, e.g., all match /16 subnets, this works out fine for constructing megaflows.
  - however, when different flows match **different subnet sizes**, like any standard IP routing table does, the constructed megaflows match the **longest subnet prefix**, e.g., any host route (/32) forces all the megaflows to match full addresses
- Example:
  - OVS is constructing a MF for a packet addressed to 10.5.6.7
  - if flows match subnet 10/8 and host 10.1.2.3/32, one could safely install a megaflow for 10.5/16 (because 10.5/16 is completely inside 10/8 and does not include 10.1.2.3)
  - but without additional optimization OVS installs 10.5.6.7/32

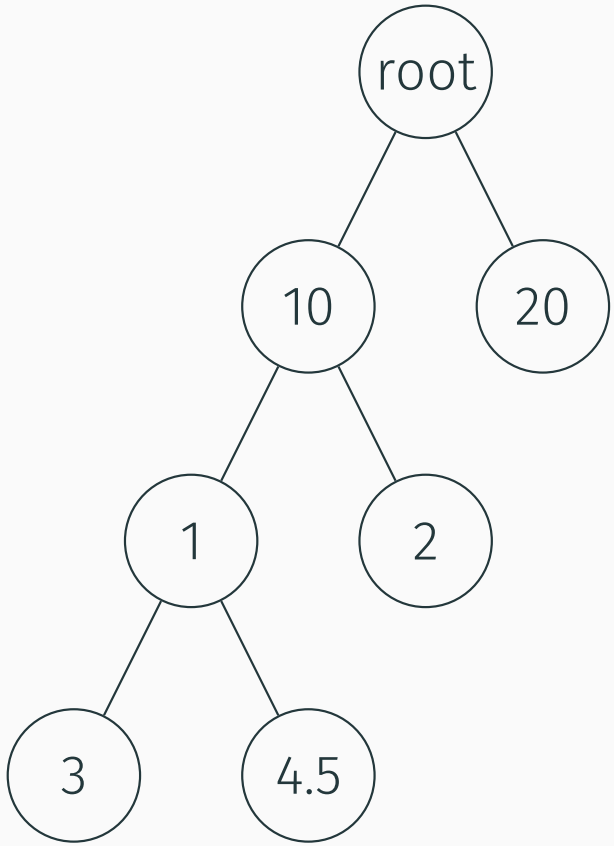
Prefix Tracking algorithm

```
function TrieSearch(value, root)
    node ← root, prev ← NULL
    plens ← bit—array of len(value) 0—bits←
    i0
    while node != NULL do←
        c0
        while c < len(node.bits) do
            if value[i] != node.bits[c] then
                return (i + 1, plens)←
            cc+1,←ii+1
        if node.n rules > 0 then
            plens[-i1←]1
        if i ≥ len(value) then
            return (i,plens)
        prev ← node
        if value[i] = 0 then
        node ← node.left
        else
            node ← node.right
    if prev != NULL and prev has at least one child then←
        ii+1
    return (i,plens)
```

Prefix Tracking algorithm: example

OpenFlow table with flows matching on some IPv4 field:

```
20 /8
10.1 /16
10.2 /16
10.1.3 /24
10.1.4.5/32
```



Classifier Partitioning

- further optimization of tuple space searches: skip tuples that cannot possibly match
- OVS partitions the classifier based on a particular metadata field. If the current value in that field does not match any value in a particular tuple, the tuple is skipped altogether.
- OVS does not have a fixed pipeline like traditional switches, but NVP often configures each lookup in the classifier as a stage in a pipeline. These stages match on a fixed number of fields, similar to a tuple. By storing a numeric indicator of the pipeline stage into a specialized metadata field, NVP provides a hint to the classifier to efficiently only look at pertinent tuples.

Cache Invalidation (1)

Why is it needed?

1. controller can change the OpenFlow flow table
2. OpenFlow also specifies changes that the switch should take on its own in reaction to various events, e.g., OpenFlow “group” behavior can depend on whether carrier is detected on a network interface
3. reconfiguration that turns features on or off, adds or removes ports, etc., can affect packet handling
4. Protocols for connectivity detection, (CFM or BFD), or for loop detection and avoidance, e.g., (Rapid) Spanning Tree Protocol, can influence behavior.
5. some OpenFlow actions and Open vSwitch extensions change behavior based on network state, e.g., based on MAC learning

Cache Invalidation (2)

Two approaches:

1. changes whose effects were too broad to precisely identify the needed changes, Open vSwitch had to examine every dat- apath flow for possible changes. Each flow had to be passed through the OpenFlow flow table in the same way as it was originally constructed
2. changes whose effects on datapath flows could be narrowed down, such as MAC learning table change - *tags*

Cache Invalidation (3): tags

- Assign a tag to each property that, if changed, could require megaflow update was given one of these tags.
- Also, each megaflow was associated with the tags for all of the properties on which its actions depended
- if MAC learned port later changed: 1. Open vSwitch added the tag to a set of tags that accumulated changes, 2. OVS scanned megaflow table (in batches) for megaflows that had at least one of the changed tags, 3. checked whether their actions needed an update.
- removed in version v. 2.0 after further optimizations in favor of always revalidating the entire datapath flow table

Cache Invalidation (3): OVS 2.0 changes

- divided userspace into multiple threads. We broke flow setup into separate threads so that it did not have to wait behind revalidation
- Datapath flow eviction, however, remained part of the single main thread and could not keep up with multiple threads setting up flow
- OVS 2.1: multiple dedicated threads for cache revalidation; kernel cache maximum size is dynamically adjusted to ensure that total revalidation time stays under 1 second, to bound the amount of time that a stale entry can stay in the cache

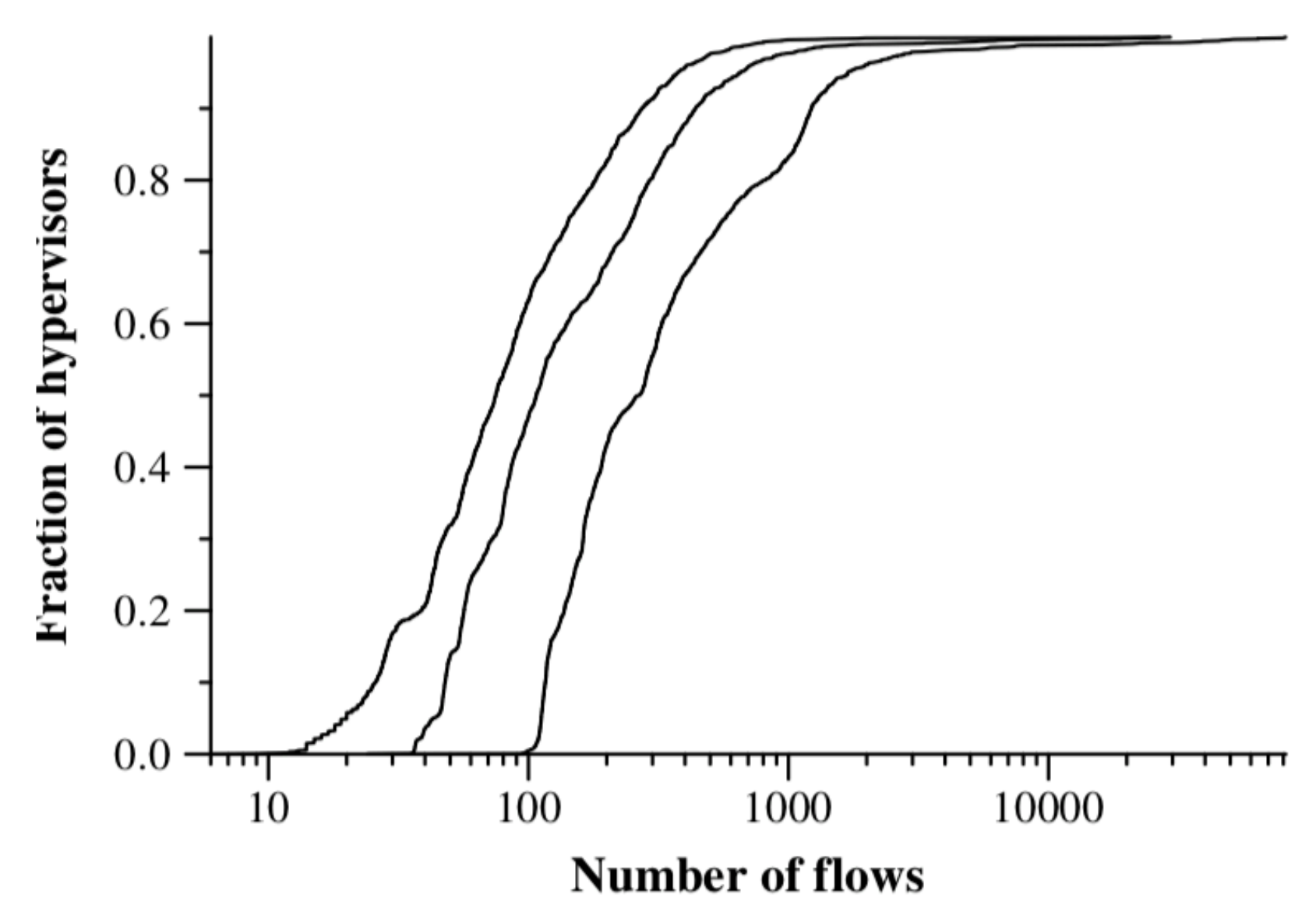
Evaluation

Performance in Production

- 24 hours of Open vSwitch performance data from the hypervisors running in Rackspace data center
- statistics polled every 10 minutes from over 1,000 hypervisors
- commercial mixed tenant workloads

Cache sizes

Min/mean/max megaflow flow counts observed:



Cache hit rates

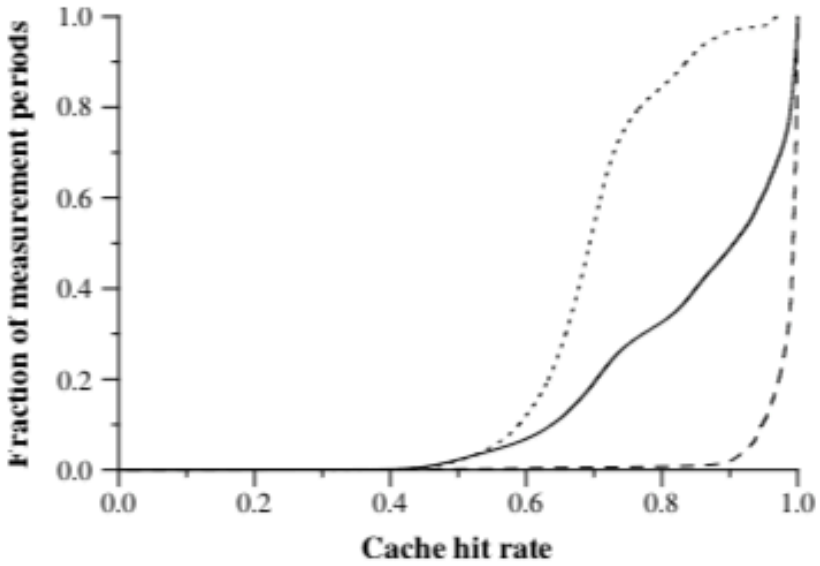


Figure 5: Hit rates during all (solid), busiest (dashed), and slowest (dotted) periods.

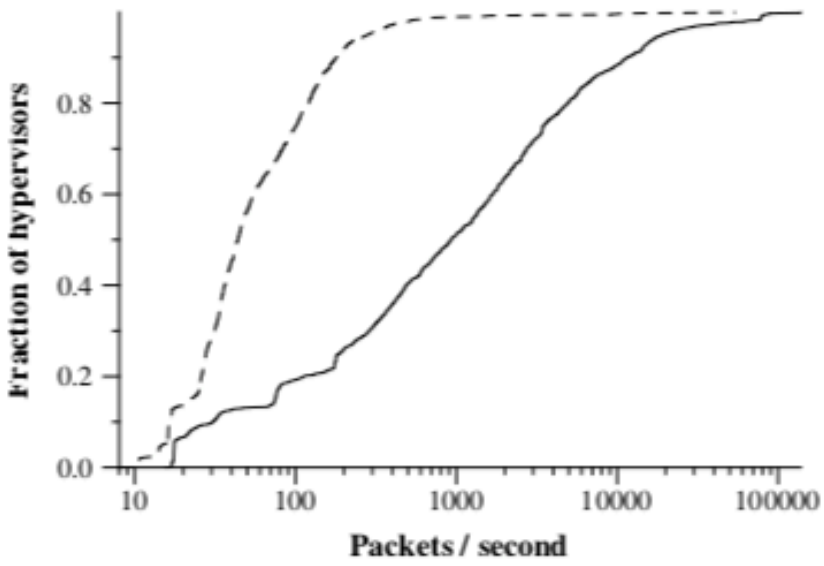


Figure 6: Cache hit (solid) and miss (dashed) packet counts.

CPU usage

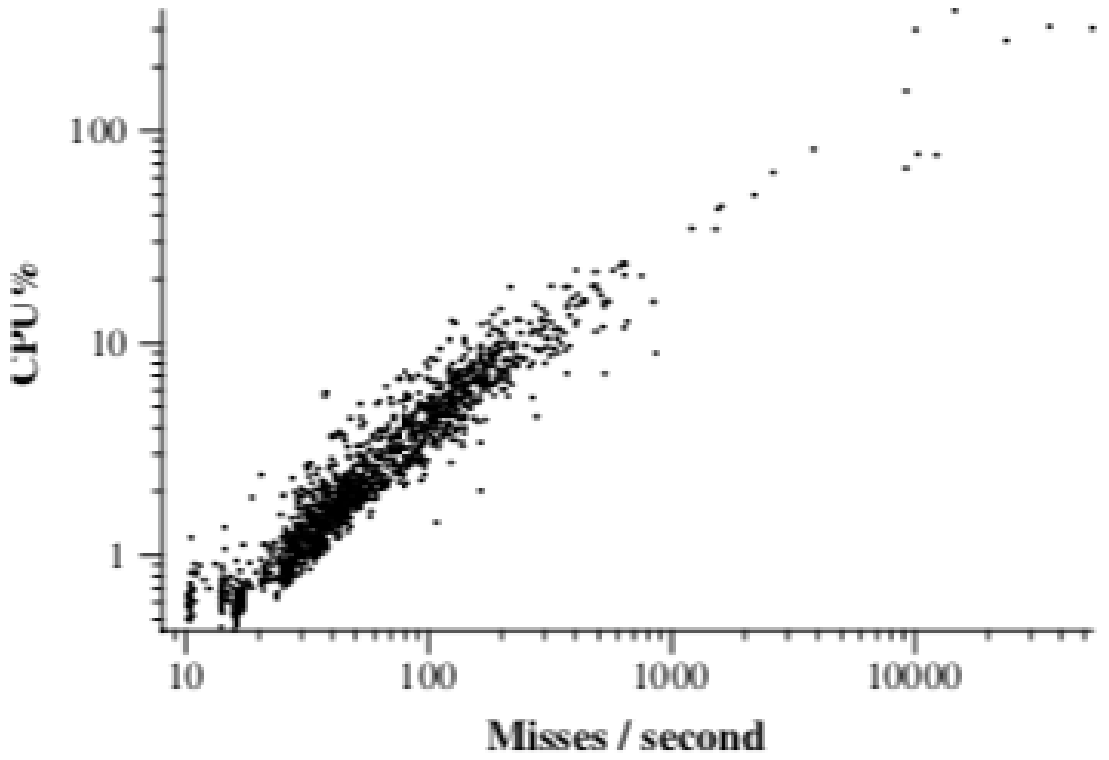


Figure 7: Userspace daemon CPU load as a function of misses/s entering userspace.

Caching Microbenchmarks: setup

Simple OpenFlow table used:

```
(1) arp
(2) ip ip dst=11.1.1.1/16
(3) tcp ip dst=9.1.1.1 tcp src=10 tcp dst=10
(4) ip ip dst=9.1.1.1/24
```

Following features can be benchmarked with it:

- megaflow caching benchmark (3)
- priority matching benchmark (2+3)
- staged lookup (3)
- prefix tracking (3+4)

Caching Microbenchmarks: setup

Simple OpenFlow table used:

```
(1) arp
(2) ip ip dst=11.1.1.1/16
(3) tcp ip dst=9.1.1.1 tcp src=10 tcp dst=10
(4) ip ip dst=9.1.1.1/24
```

Following features can be benchmarked with it:

- megaflow caching benchmark (3)
- priority matching benchmark (2+3)
- staged lookup (3)
- prefix tracking (3+4)

Caching Microbenchmarks: classifier optimizations

Optimizations	ktps	Flows	Masks	CPU%
Megaflows disabled	37	1,051,884	1	45/ 40
No optimizations	56	905,758	3	37/ 40
Priority sorting only	57	794,124	4	39/ 45
Prefix tracking only	95	13	10	0/ 15
Staged lookup only	115	14	13	0/ 15
All optimizations	117	15	14	0/ 20

Each row reports the measured number of Netperf TCP CRR transactions per second, in thousands, along with the average number of tuples searched by each packet and user and kernel CPU usage.

Caching Microbenchmarks: microflow cache

Optimizations	ktps	Flows	Masks	CPU%
Enabled	Enabled	120	1.68	0/ 20
Disabled	Enabled	92	3.21	0/ 18
Enabled	Disabled	56	1.29	38/ 40
Disabled	Disabled	56	2.45	40/ 42

Each row reports the measured number of Netperf TCP CRR transactions per second, in thousands, along with the average number of tuples searched by each packet and user and kernel CPU usage.

References

Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, and Pravin Shelar, Keith Amidon, Martín Casado.
The Design and Implementation of Open vSwitch.
Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15), 2015

P.Gupta, N.McKeown
Packet Classification Using Hierarchical Intelligent Cuttings.
In Hot Interconnects VII, pages 34–41, 1999

Venkatachary Srinivasan, Subhash Suri, George Varghese
Packet Classification Using Tuple Space Search.
SIGCOMM, 1999

Open vSwitch source code
https://github.com/openvswitch/ovs