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*[†], *Martín Casado*[*]
[*]VMware [†]Awake Networks

Operational Systems Track

Abstract

We describe the design and implementation of Open vSwitch, a multi-layer, open source virtual switch for all major hypervisor platforms. Open vSwitch was designed de novo for networking in virtual environments, resulting in major design departures from traditional software switching architectures. We detail the advanced flow classification and caching techniques that Open vSwitch uses to optimize its operations and conserve hypervisor resources. We evaluate Open vSwitch performance, drawing from our deployment experiences over the past seven years of using and improving Open vSwitch.

1 Introduction

Virtualization has changed the way we do computing over the past 15 years; for instance, many datacenters are entirely virtualized to provide quick provisioning, spillover to the cloud, and improved availability during periods of disaster recovery. While virtualization is still to reach all types of workloads, the number of virtual machines has already exceeded the number of servers and further virtualization shows no signs of stopping [1].

The rise of server virtualization has brought with it a fundamental shift in datacenter networking. A new network access layer has emerged in which most network ports are virtual, not physical [5] – and therefore, the first hop switch for workloads increasingly often resides within the hypervisor. In the early days, these hypervisor "vSwitches" were primarily concerned with providing basic network connectivity. In effect, they simply mimicked their ToR cousins by extending physical L2 networks to resident virtual machines. As virtualized workloads proliferated, limits of this approach became evident: reconfiguring and preparing a physical network for new workloads slows their provisioning, and coupling workloads with physical L2 segments severely limits their mobility and scalability to that of the underlying network.

These pressures resulted in the emergence of network virtualization [19]. In network virtualization, virtual switches become the primary provider of network services for VMs, leaving physical datacenter networks with transportation of IP tunneled packets between hypervisors. This approach allows the virtual networks to be decoupled from their underlying physical networks, and by leveraging the flexibility of general purpose processors, virtual switches can provide VMs, their tenants, and administrators with logical network abstractions, services and tools identical to dedicated physical networks.

Network virtualization demands a capable virtual switch – forwarding functionality must be wired on a per virtual port basis to match logical network abstractions configured by administrators. Implementation of these abstractions, across hypervisors, also greatly benefits from fine-grained centralized coordination. This approach starkly contrasts with early virtual switches for which a static, mostly hard-coded forwarding pipelines had been completely sufficient to provide virtual machines with L2 connectivity to physical networks.

It was this context: the increasing complexity of virtual networking, emergence of network virtualization, and limitations of existing virtual switches, that allowed Open vSwitch to quickly gain popularity. Today, on Linux, its original platform, Open vSwitch works with most hypervisors and container systems, including Xen, KVM, and Docker. Open vSwitch also works "out of the box" on the FreeBSD and NetBSD operating systems and ports to the VMware ESXi and Microsoft Hyper-V hypervisors are underway.

In this paper, we describe the design and implementation of Open vSwitch [26, 29]. The key elements of its design, revolve around the performance required by the production environments in which Open vSwitch is commonly deployed, and the programmability demanded by network virtualization. Unlike traditional network appliances, whether software or hardware, which achieve high performance through specialization, Open vSwitch, by

contrast, is designed for flexibility and general-purpose usage. It must achieve high performance without the luxury of specialization, adapting to differences in platforms supported, all while sharing resources with the hypervisor and its workloads. Therefore, this paper foremost concerns this tension – how Open vSwitch obtains high performance without sacrificing generality.

The remainder of the paper is organized as follows. Section 2 provides further background about virtualized environments while Section 3 describes the basic design of Open vSwitch. Afterward, Sections 4, 5, and 6 describe how the Open vSwitch design optimizes for the requirements of virtualized environments through flow caching, how caching has wide-reaching implications for the entire design, including its packet classifier, and how Open vSwitch manages its flow caches. Section 7 then evaluates the performance of Open vSwitch through classification and caching micro-benchmarks but also provides a view of Open vSwitch performance in a multi-tenant datacenter. Before concluding, we discuss ongoing, future and related work in Section 8.

2 Design Constraints and Rationale

The operating environment of a virtual switch is drastically different from the environment of a traditional network appliance. Below we briefly discuss constraints and challenges stemming from these differences, both to reveal the rationale behind the design choices of Open vSwitch and highlight what makes it unique.

Resource sharing. The performance goals of traditional network appliances favor designs that use dedicated hardware resources to achieve line rate performance in *worst-case* conditions. With a virtual switch on the other hand, resource conservation is critical. Whether or not the switch can keep up with worst-case line rate is secondary to maximizing resources available for the primary function of a hypervisor: running user workloads. That is, compared to physical environments, networking in virtualized environments optimizes for the *common case* over the worst-case. This is not to say worst-case situations are not important because they do arise in practice. Port scans, peer-to-peer rendezvous servers, and network monitoring all generate unusual traffic patterns but must be supported gracefully. This principle led us, *e.g.,* toward heavy use of flow caching and other forms of caching, which in common cases (with high hit rates) reduce CPU usage and increase forwarding rates.

Placement. The placement of virtual switches at the edge of the network is a source of both simplifications and complications. Arguably, topological location as a leaf, as well as sharing fate with the hypervisor and VMs remove many standard networking problems. The placement complicates scaling, however. It's not uncommon for a single virtual switch to have thousands of virtual switches as its peers in a mesh of point-to-point IP tunnels between hypervisors. Virtual switches receive forwarding state updates as VMs boot, migrate, and shut down and while virtual switches have relatively few (by networking standards) physical network ports directly attached, changes in remote hypervisors may affect local state. Especially in larger deployments of thousands (or more) of hypervisors, the forwarding state may be in constant flux. The prime example of a design influenced by this principle discussed in this paper is the Open vSwitch classification algorithm, which is designed for $O(1)$ updates.

SDN, use cases, and ecosystem. Open vSwitch has three additional unique requirements that eventually caused its design to differ from the other virtual switches:

First, Open vSwitch has been an *OpenFlow switch* since its inception. It is deliberately not tied to a single-purpose, tightly vertically integrated network control stack, but instead is re-programmable through OpenFlow [27]. This constrasts with a *feature datapath* model of other virtual switches [24, 39]: similar to forwarding ASICs, their packet processing pipelines are fixed. Only configuration of prearranged features is possible. (The Hyper-V virtual switch [24] can be extended by adding binary modules, but ordinarily each module only adds another single-purpose feature to the datapath.)

The flexibility of OpenFlow was essential in the early days of SDN but it quickly became evident that advanced use cases, such as network virtualization, result in long packet processing pipelines, and thus higher classification load than traditionally seen in virtual switches. To prevent Open vSwitch from consuming more hypervisor resources than competitive virtual switches, it was forced to implement flow caching.

Third, unlike any other major virtual switch, Open vSwitch is open source and multi-platform. In contrast to closed source virtual switches which all operate in a single environment, Open vSwitch's environment is usually selected by a user who chooses an operating system distribution and hypervisor. This has forced the Open vSwitch design to be quite modular and portable.

3 Design

3.1 Overview

In Open vSwitch, two major components direct packet forwarding. The first, and larger, component is `ovs-vswitchd`, a userspace daemon that is essentially the same from one operating system and operating environment to another. The other major component, a

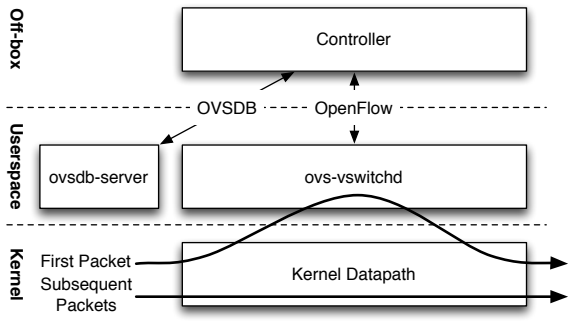


Figure 1: The components and interfaces of Open vSwitch. The first packet of a flow results in a miss, and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel.

datapath kernel module, is usually written specially for the host operating system for performance.

Figure 1 depicts how the two main OVS components work together to forward packets. The datapath module in the kernel receives the packets first, from a physical NIC or a VM's virtual NIC. Either `ovs-vswitchd` has instructed the datapath how to handle packets of this type, or it has not. In the former case, the datapath module simply follows the instructions, called *actions*, given by `ovs-vswitchd`, which list physical ports or tunnels on which to transmit the packet. Actions may also specify packet modifications, packet sampling, or instructions to drop the packet. In the other case, where the datapath has not been told what to do with the packet, it delivers it to `ovs-vswitchd`. In userspace, `ovs-vswitchd` determines how the packet should be handled, then it passes the packet back to the datapath with the desired handling. Usually, `ovs-vswitchd` also tells the datapath to cache the actions, for handling similar future packets.

In Open vSwitch, flow caching has greatly evolved over time; the initial datapath was a *microflow cache*, essentially caching per transport connection forwarding decisions. In later versions, the datapath has two layers of caching: a microflow cache and a secondary layer, called a *megaflow cache*, which caches forwarding decisions for traffic aggregates beyond individual connections. We will return to the topic of caching in more detail in Section 4.

Open vSwitch is commonly used as an SDN switch, and the main way to control forwarding is OpenFlow [27]. Through a simple binary protocol, OpenFlow allows a controller to add, remove, update, monitor, and obtain statistics on flow tables and their flows, as well as to divert selected packets to the controller and to inject packets from the controller into the switch. In Open vSwitch, `ovs-vswitchd` receives OpenFlow flow tables from an SDN controller, matches any packets received from the datapath module against these OpenFlow tables, gathers the actions applied, and finally caches the result in the kernel datapath. This allows the datapath module to remain unaware of the particulars of the OpenFlow wire protocol, further simplifying it. From the OpenFlow controller's point of view, the caching and separation into user and kernel components are invisible implementation details: in the controller's view, each packet visits a series of OpenFlow flow tables and the switch finds the highest-priority flow whose conditions are satisfied by the packet, and executes its OpenFlow actions.

The flow programming model of Open vSwitch largely determines the use cases it can support and to this end, Open vSwitch has many extensions to standard OpenFlow to accommodate network virtualization. We will discuss these extensions shortly, but before that, we turn our focus on the performance critical aspects of this design: packet classification and the kernel-userspace interface.

3.2 Packet Classification

Algorithmic packet classification is expensive on general purpose processors, and packet classification in the context of OpenFlow is especially costly because of the generality of the form of the match, which may test any combination of Ethernet addresses, IPv4 and IPv6 addresses, TCP and UDP ports, and many other fields, including packet metadata such as the switch ingress port.

Open vSwitch uses a *tuple space search* classifier [34] for all of its packet classification, both kernel and userspace. To understand how tuple space search works, assume that all the flows in an Open vSwitch flow table matched on the same fields in the same way, *e.g.,* all flows match the source and destination Ethernet address but no other fields. A tuple search classifier implements such a flow table as a single hash table. If the controller then adds new flows with a different form of match, the classifier creates a second hash table that hashes on the fields matched in those flows. (The *tuple* of a hash table in a tuple space search classifier is, properly, the set of fields that form that hash table's key, but we often refer to the hash table itself as the tuple, as a kind of useful shorthand.) With two hash tables, a search must look in both hash tables. If there are no matches, the flow table doesn't contain a match; if there is a match in one hash table, that flow is the result; if there is a match in both, then the result is the flow with the higher priority. As the controller continues to add more flows with new forms of match, the classifier similarly expands to include a hash table for each unique match, and a search of the classifier must look in every hash table.

While the lookup complexity of tuple space search is far from the state of the art [8, 18, 38], it performs well with the flow tables we see in practice and has three attractive properties over decision tree classification algorithms. First, it supports efficient constant-time updates (an up-

date translates to a single hash table operation), which makes it suitable for use with virtualized environments where a centralized controller may add and remove flows often, sometimes multiple times per second per hypervisor, in response to changes in the whole datacenter. Second, tuple space search generalizes to an arbitrary number of packet header fields, without any algorithmic change. Finally, tuple space search uses memory linear in the number of flows.

The relative cost of a packet classification is further amplified by the large number of flow tables that sophisticated SDN controllers use. For example, flow tables installed by the VMware network virtualization controller [19] use a minimum of about 15 table lookups per packet in its packet processing pipeline. Long pipelines are driven by two factors: reducing stages through cross-producting would often significantly increase the flow table sizes and developer preference to modularize the pipeline design. Thus, even more important than the performance of a single classifier lookup, it is to reduce the number of flow table lookups a single packet requires, on average.

3.3 OpenFlow as a Programming Model

Initially, Open vSwitch focused on a reactive flow programming model in which a controller responding to traffic installs microflows which match every supported OpenFlow field. This approach is easy to support for software switches and controllers alike, and early research suggested it was sufficient [3]. However, reactive programming of microflows soon proved impractical for use outside of small deployments and Open vSwitch had to adapt to proactive flow programming to limit its performance costs.

In OpenFlow 1.0, a microflow has about 275 bits of information, so that a flow table for every microflow would have 2^{275} or more entries. Thus, proactive population of flow tables requires support for wildcard matching to cover the header space of all possible packets. With a single table this results in a "cross-product problem": to vary the treatment of packets according to n_1 values of field A and n_2 values of field B, one must install $n_1 \times n_2$ flows in the general case, even if the actions to be taken based on A and B are independent. Open vSwitch soon introduced an extension action called *resubmit* that allows packets to consult multiple flow tables (or the same table multiple times), aggregating the resulting actions. This solves the cross-product problem, since one table can contain n_1 flows that consult A and another table n_2 flows that consult B. The resubmit action also enables a form of programming based on multiway branching based on the value of one or more fields. Later, OpenFlow vendors focusing on hardware sought a way to make better use of the multiple tables consulted in series by forwarding ASICs, and OpenFlow 1.1 introduced multi-table support. Open vSwitch adopted the new model but retained its support for the resubmit action for backward compatibility and because the new model did not allow for recursion but only forward progress through a fixed table pipeline.

At this point, a controller could implement programs in Open vSwitch flow tables that could make decisions based on packet headers using arbitrary chains of logic, but they had no access to temporary storage. To solve that problem, Open vSwitch extended OpenFlow in another way, by adding meta-data fields called "registers" that flow tables could match, plus additional actions to modify and copy them around. With this, for instance, flows could decide a physical destination early in the pipeline, then run the packet through packet processing steps identical regardless of the chosen destination, until sending the packet, possibly using destination-specific instructions. As another example, VMware's NVP network virtualization controller [19] uses registers to keep track of a packet's progress through a logical L2 and L3 topology implemented as "logical datapaths" that it overlays on the physical OpenFlow pipeline.

OpenFlow is specialized for flow-based control of a switch. It cannot create or destroy OpenFlow switches, add or remove ports, configure QoS queues, associate OpenFlow controller and switches, enable or disable STP (Spanning Tree Protocol), etc. In Open vSwitch, this functionality is controlled through a separate component, the *configuration database*. To access the configuration database, an SDN controller may connect to `ovsdb-server` over the OVSDB protocol [28], as shown in Figure 1. In general, in Open vSwitch, OpenFlow controls potentially fast-changing and ephemeral data such as the flow table, whereas the configuration database contains more durable state.

4 Flow Cache Design

This section describes the design of flow caching in Open vSwitch and how it evolved to its current state.

4.1 Microflow Caching

In 2007, when the development of the code that would become Open vSwitch started on Linux, only in-kernel packet forwarding could realistically achieve good performance, so the initial implementation put all OpenFlow processing into a kernel module. The module received a packet from a NIC or VM, classified through the OpenFlow table (with standard OpenFlow matches and actions), modified it as necessary, and finally sent it to another port. This approach soon became impractical because of the relative difficulty of developing in the kernel and distribut-

ing and updating kernel modules. It also became clear that an in-kernel OpenFlow implementation would not be acceptable as a contribution to upstream Linux, which is an important requirement for mainstream acceptance for software with kernel components.

Our solution was to reimplement the kernel module as a *microflow cache* in which a single cache entry exact matches with all the packet header fields supported by OpenFlow. This allowed radical simplification, by implementing the kernel module as a simple hash table rather than as a complicated, generic packet classifier, supporting arbitrary fields and masking. In this design, cache entries are extremely fine-grained and match *at most* packets of a single transport connection: even for a single transport connection, a change in network path and hence in IP TTL field would result in a miss, and would divert a packet to userspace, which consulted the actual OpenFlow flow table to decide how to forward it. This implies that the critical performance dimension is flow setup time, the time that it takes for the kernel to report a microflow "miss" to userspace and for userspace to reply.

Over multiple Open vSwitch versions, we adopted several techniques to reduce flow setup time with the microflow cache. Batching flow setups that arrive together improved flow setup performance about 24%, for example, by reducing the average number of system calls required to set up a given microflow. Eventually, we also distributed flow setup load over multiple userspace threads to benefit from multiple CPU cores. Drawing inspiration from CuckooSwitch [42], we adopted optimistic concurrent cuckoo hashing [6] and RCU [23] techniques to implement nonblocking multiple-reader, single-writer flow tables.

After general optimizations of this kind customer feedback drew us to focus on performance in latency-sensitive applications, and that required us to reconsider our simple caching design.

4.2 Megaflow Caching

While the microflow cache works well with most traffic patterns, it suffers serious performance degradation when faced with large numbers of short lived connections. In this case, many packets miss the cache, and must not only cross the kernel-userspace boundary, but also execute a long series of expensive packet classifications. While batching and multithreading can somewhat alleviate this stress, they are not sufficient to fully support this workload.

We replaced the microflow cache with a *megaflow cache*. The megaflow cache is a single flow lookup table that supports generic matching, *i.e.,* it supports caching forwarding decisions for larger aggregates of traffic than connections. While it more closely resembles a generic OpenFlow table than the microflow cache does, due to its support for arbitrary packet field matching, it is still strictly simpler and lighter in runtime for two primary reasons. First, it does not have priorities, which speeds up packet classification: the in-kernel tuple space search implementation can terminate as soon as it finds any match, instead of continuing to look for a higher-priority match until all the mask-specific hash tables are inspected. (To avoid ambiguity, userspace installs only disjoint megaflows, those whose matches do not overlap.) Second, there is only one megaflow classifier, instead of a pipeline of them, so userspace installs megaflow entries that collapse together the behavior of all relevant OpenFlow tables.

The cost of a megaflow lookup is close to the general-purpose packet classifier, even though it lacks support for flow priorities. Searching the megaflow classifier requires searching each of its hash tables until a match is found; and as discussed in Section 3.2, each unique kind of match in a flow table yields a hash table in the classifier. Assuming that each hash table is equally likely to contain a match, matching packets require searching $(n+1)/2$ tables on average, and non-matching packets require searching all n. Therefore, for $n > 1$, which is usually the case, a classifier-based megaflow search requires more hash table lookups than a microflow cache. Megaflows by themselves thus yield a trade-off: one must bet that the per-microflow benefit of avoiding an extra trip to userspace outweighs the per-packet cost of the extra hash lookups in form of megaflow lookup.

Open vSwitch addresses the costs of megaflows by retaining the microflow cache as a first-level cache, consulted before the megaflow cache. This cache is a hash table that maps from a microflow to its matching megaflow. Thus, after the first packet in a microflow passes through the kernel megaflow table, requiring a search of the kernel classifier, this exact-match cache allows subsequent packets in the same microflow to get quickly directed to the appropriate megaflow. This reduces the cost of megaflows from per-packet to per-microflow. The exact-match cache is a true cache in that its activity is not visible to userspace, other than through its effects on performance.

A megaflow flow table represents an active subset of the cross-product of all the userspace OpenFlow flow tables. To avoid the cost of proactive crossproduct computation and to populate the megaflow cache only with entries relevant for current forwarded traffic, the Open vSwitch userspace daemon computes the cache entries incrementally and reactively. As Open vSwitch processes a packet through userspace flow tables, classifying the packet at every table, it tracks the packet field bits that were consulted as part of the classification algorithm. The generated megaflow must match any field (or part of a field) whose value was used as part of the decision. For

example, if the classifier looks at the IP destination field in any OpenFlow table as part of its pipeline, then the megaflow cache entry's condition must match on the destination IP as well. This means that incoming packets drive the cache population, and as the aggregates of the traffic evolve, new entries are populated and old entries removed.

The foregoing discussion glosses over some details. The basic algorithm, while correct, produces match conditions that are more specific than necessary, which translates to suboptimal cache hit rates. Section 5, below, describes how Open vSwitch modifies tuple space search to yield better megaflows for caching. Afterward, Section 6 addresses cache invalidation.

5 Caching-aware Packet Classification

We now turn our focus on the refinements and improvements we made to the basic tuple search algorithm (summarized in Section 3.2) to improve its suitability for flow caching.

5.1 Problem

As Open vSwitch userspace processes a packet through its OpenFlow tables, it tracks the packet field bits that were consulted as part of the forwarding decision. This bitwise tracking of packet header fields is very effective in constructing the megaflow entries with simple OpenFlow flow tables.

For example, if the OpenFlow table only looks at Ethernet addresses (as would a flow table based on L2 MAC learning), then the megaflows it generates will also look only at Ethernet addresses. For example, port scans (which do not vary Ethernet addresses) will not cause packets to go to userspace as their L3 and L4 header fields will be wildcarded resulting in near-ideal megaflow cache hit rates. On the other hand, if even one flow entry in the table matches on the TCP destination port, tuple space search will consider the TCP destination port of every packet. Then every megaflow will also match on the TCP destination port, and port scan performance again drops.

We do not know of an efficient online algorithm to generate optimal, least specific megaflows, so in development we have focused our attention on generating increasingly good approximations. Failing to match a field that must be included can cause incorrect packet forwarding, which makes such errors unacceptable, so our approximations are biased toward matching on more fields than necessary. The following sections describe improvements of this type that we have integrated into Open vSwitch.

```
function PRIORITYSORTEDTUPLESEARCH(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
```

Figure 2: Tuple space search for target packet headers H, with priority sorting.

5.2 Tuple Priority Sorting

Lookup in a tuple space search classifier ordinarily requires searching every tuple. Even if a search of an early tuple finds a match, the search must still look in the other tuples because one of them might contain a matching flow with a higher priority.

We improved on this by tracking, in each tuple T, the maximum priority $T.pri\_max$ of any flow entry in T. We modified the lookup code to search tuples from greatest to least maximum priority, so that a search that finds a matching flow F with priority $F.pri$ can terminate as soon as it arrives at a tuple whose maximum priority is $F.pri$ or less, since at that point no better match can be found. Figure 2 shows the algorithm in detail.

As an example, we examined the OpenFlow table installed by a production deployment of VMware's NVP controller [19]. This table contained 29 tuples. Of those 29 tuples, 26 contained flows of a single priority, which makes intuitive sense because flows matching a single tuple tend to share a purpose and therefore a 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. The final tuple contained 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.

5.3 Staged Lookup

Tuple space search searches each tuple with a hash table lookup. In our algorithm to construct the megaflow matching condition, this hash table lookup means that the megaflow must match all the bits of fields included in the tuple, even if the tuple search fails, because every one of those fields and their bits may have affected the

lookup result so far. When the tuple matches on a field that varies often from flow to flow, *e.g.,* the TCP source port, the generated megaflow is not much more useful than installing a microflow would be because it will only match a single TCP stream.

This points to an opportunity for improvement. If one could search a tuple on a subset of its fields, and determine with this search that the tuple could not possibly match, then the generated megaflow would only need to match on the subset of fields, rather than all the fields in the tuple.

The tuple implementation as a hash table over all its fields made such an optimization difficult. One cannot search a hash table on a subset of its key. We considered other data structures. 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 $O(1)$ to $O(n)$ in the length of the tuple fields. Individual per-field hash tables had the same drawback. We did not consider data structures larger than $O(n)$ in the number of flows in a tuple, because OpenFlow tables can have hundreds of thousands of flows.

The solution we implemented statically divides fields into four groups, in decreasing order of traffic granularity: metadata (*e.g.,* the switch ingress port), L2, L3, and L4. We changed each tuple from a single hash table to an array of four hash tables, called *stages*: one over metadata fields only, one over metadata and L2 fields, one over metadata, L2, and L3 fields, and one over all fields. (The latter is the same as the single hash table in the previous implementation.) A 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 the megaflow match.

This optimization technique would apply to any subsets of the supported fields, not just the layer-based subsets we used. We divided fields by protocol layer because, as a rule of thumb, in TCP/IP, inner layer headers tend to be more diverse than outer layer headers. At L4, for example, the TCP source and destination ports change on a per-connection basis, but in the metadata layer only a relatively small and static number of ingress ports exist.

Each stage in a tuple includes all of the fields in earlier stages. We chose this arrangement, although the technique does not require it, because then hashes could be computed incrementally from one stage to the next, and profiling had shown hash computation to be a significant cost (with or without staging).

With four stages, one might expect the time to search a tuple to quadruple. Our measurements show that, in fact, classification speed actually improves slightly in practice because, when a search terminates at any early stage, the classifier does not have to compute the full hash of all the fields covered by the tuple.

This optimization fixes a performance problem observed in production deployments. The NVP controller uses Open vSwitch to implement multiple isolated logical datapaths (further interconnected to form logical networks). Each logical datapath is independently configured. Suppose that some logical datapaths are configured with ACLs that allow or deny traffic based on L4 (*e.g.,* TCP or UDP) port numbers. Megaflows for traffic on these logical datapaths must match on the L4 port to enforce the ACLs. Megaflows for traffic on other logical datapaths need not and, for performance, should not match on L4 port. Before this optimization, however, all generated megaflows matched on L4 port because a classifier search had to pass through a tuple that matched on L4 port. The optimization allows megaflows for traffic on logical datapaths without L4 ACLs to avoid matching on L4 port, because the first three (or fewer) stages are enough to determine that there is no match.

5.4 Prefix Tracking

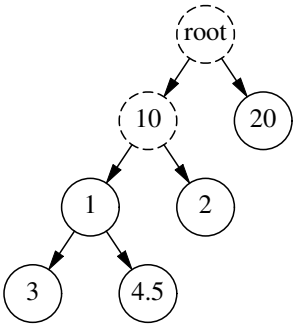
Flows in OpenFlow often match IPv4 and IPv6 subnets to implement routing. When all the flows that match on such a field use the same subnet size, *e.g.,* all match /16 subnets, this works out fine for constructing megaflows. If, on the other hand, different flows match different subnet sizes, like any 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. Suppose, for example, Open vSwitch is constructing a megaflow 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 Open vSwitch installs 10.5.6.7/32. (Our examples use only octet prefixes, *e.g.,* /8, /16, /24, /32, for clarity, but the implementation and the pseudocode shown later work in terms of bit prefixes.)

We implemented optimization of prefixes for IPv4 and IPv6 fields using a trie structure. If a flow table matches over an IP address, the classifier executes an LPM lookup for any such field *before* the tuple space search, both to determine the maximum megaflow prefix length required, as well as to determine which tuples can be skipped entirely without affecting correctness.[1] As an example, suppose an OpenFlow table contained flows that matched on some IPv4 field, as shown:

[1] This is a slight simplification for improved clarity; the actual implementation reverts to prefix tracking if staged lookups have concluded to include an IP field to the match.

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

These flows correspond to the following trie, in which a solid circle represents one of the address matches listed above and a dashed circle indicates a node that is present only for its children:



To determine the bits to match, Open vSwitch traverses the trie from the root down through nodes with labels matching the corresponding bits in the packet's IP address. If traversal reaches a leaf node, then the megaflow need not match the remainder of the address bits, *e.g.,* in our example 10.1.3.5 would be installed as 10.1.3/24 and 20.0.5.1 as 20/8. If, on the other hand, traversal stops due to the bits in the address not matching any of the corresponding labels in the tree, the megaflow must be constructed to match up to and including the bits that could not be found, *e.g.,* 10.3.5.1 must be installed as 10.3/16 and 30.10.5.2 as 30/8.

The trie search result also allows Open vSwitch to skip searching some tuples. Consider the address 10.1.6.1. A search of the above trie for this address terminates at the node labeled 1, failing to find a node to follow for the address's third octet. This means that no flow in the flow table with an IP address match longer than 16 bits matches the packet, so the classifier lookup can skip searching tuples for the flows listed above with /24 and /32 prefixes.

Figure 3 gives detailed pseudocode for the prefix matching algorithm. Each node is assumed to have members *bits*, the bits in the particular node (at least one bit, except that the root node may be empty); *left* and *right*, the node's children (or NULL); and *n_rules*, the number of rules in the node (zero if the node is present only for its children, otherwise nonzero). It returns the number of bits that must be matched, allowing megaflows to be improved, and a bit-array in which 0-bits designate matching lengths for tuples that Open vSwitch may skip searching, as described above.

While this algorithm optimizes longest-prefix match lookups, it improves megaflows even when no flow explicitly matches against an IP prefix. To implement a longest prefix match in OpenFlow, the flows with longer prefix must have higher priorities, which will allow the tuple priority sorting optimization in Section 5.2 to skip prefix matching tables after the longest match is found, but this alone causes megaflows to unwildcard address bits according to the longest prefix in the table. The main practical benefit of this algorithm, then, is to prevent policies (such as a high priority ACL) that are applied to a specific host from forcing all megaflows to match on a full IP address. This algorithm allows the megaflow entries only to match with the high order bits sufficient to differentiate the traffic from the host with ACLs.

```
function TRIESEARCH(value, root)
    node ← root, prev ← NULL
    plens ← bit-array of len(value) 0-bits
    i ← 0
    while node ≠ NULL do
        c ← 0
        while c < len(node.bits) do
            if value[i] ≠ node.bits[c] then
                return (i+1, plens)
            c ← c+1, i ← i+1
        if node.n_rules > 0 then
            plens[i−1] ← 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
        i ← i+1
    return (i, plens)
```

Figure 3: Prefix tracking pseudocode. The function searches for *value* (*e.g.,* an IP address) in the trie rooted at node *root*. It returns the number of bits at the beginning of *value* that must be examined to render its matching node unique, and a bit-array of possible matching lengths. In the pseudocode, $x[i]$ is bit i in x and $\text{len}(x)$ the number of bits in x.

We also eventually adopted prefix tracking for L4 transport port numbers. Similar to IP ACLs, this prevents high-priority ACLs that match specific transport ports (*e.g.,* to block SMTP) from forcing all megaflows to match the entire transport port fields, which would again reduce the megaflow cache to a microflow cache [32].

5.5 Classifier Partitioning

The number of tuple space searches can be further reduced by skipping tuples that cannot possibly match. OpenFlow

supports setting and matching metadata fields during a packet's trip through the classifier. Open vSwitch 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.

While Open vSwitch does not have a fixed pipeline like traditional switches, 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.

6 Cache Invalidation

The flip side of caching is the complexity of managing the cache. In Open vSwitch, the cache may require updating for a number of reasons. Most obviously, the controller can change the OpenFlow flow table. 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. Reconfiguration that turns features on or off, adds or removes ports, etc., can affect packet handling. Protocols for connectivity detection, such as CFM [10] or BFD [14], or for loop detection and avoidance, *e.g.,* (Rapid) Spanning Tree Protocol, can influence behavior. Finally, some OpenFlow actions and Open vSwitch extensions change behavior based on network state, *e.g.,* based on MAC learning.

Ideally, Open vSwitch could precisely identify the megaflows that need to change in response to some event. For some kinds of events, this is straightforward. For example, when the Open vSwitch implementation of MAC learning detects that a MAC address has moved from one port to another, the datapath flows that used that MAC are the ones that need an update. But the generality of the OpenFlow model makes precise identification difficult in other cases. One example is adding a new flow to an OpenFlow table. Any megaflow that matched a flow in that OpenFlow table whose priority is less than the new flow's priority should potentially now exhibit different behavior, but we do not know how to efficiently (in time and space) identify precisely those flows.[2] The problem is worsened further by long sequences of OpenFlow flow table lookups. We concluded that precision is not practical in the general case.

Therefore, early versions of Open vSwitch divided changes that could require the behavior of datapath flows to change into two groups. For the first group, the changes whose effects were too broad to precisely identify the needed changes, Open vSwitch had to examine every datapath flow for possible changes. Each flow had to be passed through the OpenFlow flow table in the same way as it was originally constructed, then the generated actions compared against the ones currently installed in the datapath. This can be time-consuming if there are many datapath flows, but we have not observed this to be a problem in practice, perhaps because there are only large numbers of datapath flows when the system actually has a high network load, making it reasonable to use more CPU on networking. The real problem was that, because Open vSwitch was single-threaded, the time spent re-examining all of the datapath flows blocked setting up new flows for arriving packets that did not match any existing datapath flow. This added high latency to flow setup for those packets, greatly increased the overall variability of flow setup latency, and limited the overall flow setup rate. Through version 2.0, therefore, Open vSwitch limited the maximum number of cached flows installed in the datapath to about 1,000, increased to 2,500 following some optimizations, to minimize these problems.

The second group consisted of changes whose effects on datapath flows could be narrowed down, such as MAC learning table changes. Early versions of Open vSwitch implemented these in an optimized way using a technique called *tags*. Each property that, if changed, could require megaflow updates was given one of these tags. Also, each megaflow was associated with the tags for all of the properties on which its actions depended, *e.g.,* if the actions output the packet to port x because the packet's destination MAC was learned to be on that port, then the megaflow is associated with the tag for that learned fact. Later, if that MAC learned port changed, Open vSwitch added the tag to a set of tags that accumulated changes. In batches, Open vSwitch scanned the megaflow table for megaflows that had at least one of the changed tags, and checked whether their actions needed an update.

Over time, as controllers grew more sophisticated and flow tables more complicated, and as Open vSwitch added more actions whose behavior changed based on network state, each datapath flow became marked with more and more tags. We had implemented tags as Bloom filters [2], which meant that each additional tag caused more "false positives" for revalidation, so now most or all flows required examination whenever any state changed. By Open vSwitch version 2.0, the effectiveness of tags had declined so much that to simplify the code Open vSwitch abandoned them altogether in favor of always revalidating the entire datapath flow table.

Since tags had been one of the ways we sought to minimize flow setup latency, we now looked for other ways. In Open vSwitch 2.0, toward that purpose, we divided userspace into multiple threads. We broke flow setup into separate threads so that it did not have to wait behind

[2] Header space analysis [16] provides the algebra to identify the flows but the feasibility of efficient, online analysis (such as in [15]) in this context remains an open question.

revalidation. Datapath flow eviction, however, remained part of the single main thread and could not keep up with multiple threads setting up flows. Under heavy flow setup load, though, the rate at which eviction can occur is critical, because userspace must be able to delete flows from the datapath as quickly as it can install new flows, or the datapath cache will quickly fill up. Therefore, in Open vSwitch 2.1 we introduced multiple dedicated threads for cache revalidation, which allowed us to scale up the revalidation performance to match the flow setup performance and to greatly increase the kernel cache maximum size, to about 200,000 entries. The actual maximum 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.

Open vSwitch userspace obtains datapath cache statistics by periodically (about once per second) polling the kernel module for every flow's packet and byte counters. The core use of datapath flow statistics is to determine which datapath flows are useful and should remain installed in the kernel and which ones are not processing a significant number of packets and should be evicted. Short of the table's maximum size, flows remain in the datapath until they have been idle for a configurable amount of time, which now defaults to 10 s. (Above the maximum size, Open vSwitch drops this idle time to force the table to shrink.) The threads that periodically poll the kernel for per flow statistics also use those statistics to implement OpenFlow's per-flow packet and byte count statistics and flow idle timeout features. This means that OpenFlow statistics are themselves only periodically updated.

The above describes how userspace invalidates the datapath's megaflow cache. Maintenance of the first-level microflow cache (discussed in Section 4) is much simpler. A microflow cache entry is only a hint to the first hash table to search in the general tuple space search. Therefore, a stale microflow cache entry is detected and corrected the first time a packet matches it. The microflow cache has a fixed maximum size, with new microflows replacing old ones, so there is no need to periodically flush old entries. We use a pseudo-random replacement policy, for simplicity, and have found it to be effective in practice.

7 Evaluation

The following sections examine Open vSwitch performance in production and in microbenchmarks.

7.1 Performance in Production

We examined 24 hours of Open vSwitch performance data from the hypervisors in a large, commercial multi-tenant data center operated by Rackspace. Our data set contains statistics polled every 10 minutes from over 1,000 hypervisors running Open vSwitch to serve mixed tenant workloads in network virtualization setting.

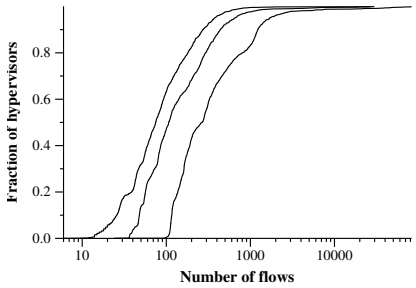


Figure 4: Min/mean/max megaflow flow counts observed.

Cache sizes. The number of active megaflows gives us an indication about practical megaflow cache sizes Open vSwitch handles. In Figure 4, we show the CDF for minimum, mean and maximum counts during the observation period. The plots show that small megaflow caches are sufficient in practice: 50% of the hypervisors had mean flow counts of 107 or less. The 99th percentile of the maximum flows was still just 7,033 flows. For the hypervisors in this environment, Open vSwitch userspace can maintain a sufficiently large kernel cache. (With the latest Open vSwitch mainstream version, the kernel flow limit is set to 200,000 entries.)

Cache hit rates. Figure 5 shows the effectiveness of caching. The solid line plots the overall cache hit rate across each of the 10-minute measurement intervals across the entire population of hypervisors. The overall cache hit rate was 97.7%. The dotted line includes just the 25% of the measurement periods in which the fewest packets were forwarded, in which the caching was less effective than overall, achieving a 74.7% hit rate. Intuitively, caching is less effective (and unimportant) when there is little to cache. Open vSwitch caching is most effective when it is most useful: when there is a great deal of traffic to cache. The dashed line, which includes just the 25% of the measurement periods in which the most packets were forwarded, demonstrates this: during these periods, the hit rate rises slightly above the overall average to 98.0%.

The vast majority of the hypervisors in this data center do not experience high volume traffic from their workloads. Figure 6 depicts this: 99% of the hypervisors see fewer than 79,000 packets/s to hit their caches (and fewer than 1500 flow setups/s to enter userspace due to misses).

CPU usage. Our statistics gathering process cannot separate Open vSwitch kernel load from the rest of the kernel load, so we focus on Open vSwitch userspace. As we

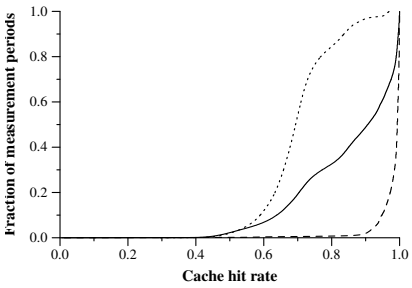


Figure 5: Hit rates during all (solid), busiest (dashed), and slowest (dotted) periods.

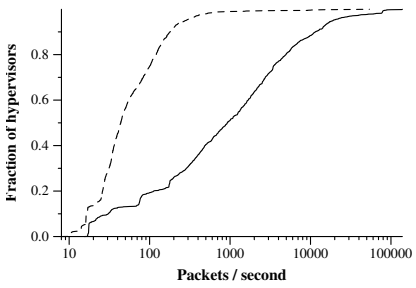


Figure 6: Cache hit (solid) and miss (dashed) packet counts.

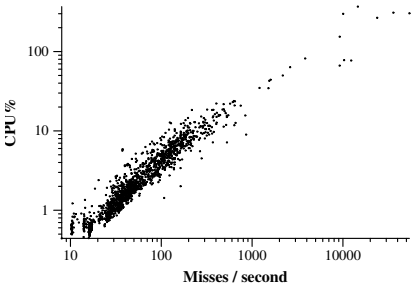


Figure 7: Userspace daemon CPU load as a function of misses/s entering userspace.

will show in Section 7.2, the megaflow CPU usage itself is in line with Linux bridging and less of a concern. In Open vSwitch, the userspace load is largely due to the misses in kernel and Figure 7 depicts this. (Userspace CPU load can exceed 100% due to multithreading.) We observe that 80% of the hypervisors averaged 5% CPU or less on `ovs-vswitchd`, which has been our traditional goal. Over 50% of hypervisors used 2% CPU or less.

Outliers. The upper right corner of Figure 7 depicts a number of hypervisors using large amounts of CPU to process many misses in userspace. We individually examined the six most extreme cases, where Open vSwitch averaged over 100% CPU over the 24 hour period. We found that all of these hypervisors exhibited a previously unknown bug in the implementation of prefix tracking, such that flows that match on an ICMP type or code caused all TCP flows to match on the entire TCP source or destination port, respectively. We believe we have fixed this bug in Open vSwitch 2.3, but the data center was not upgraded in time to verify in production.

7.2 Caching Microbenchmarks

We ran microbenchmarks with a simple flow table designed to compactly demonstrate the benefits of the caching-aware packet classification algorithm. We used the following OpenFlow flows, from highest to lowest priority. We omit the actions because they are not significant for the discussion:

```
arp                                                    (1)
ip   ip_dst=11.1.1.1/16                                (2)
tcp  ip_dst=9.1.1.1      tcp_src=10 tcp_dst=10         (3)
ip   ip_dst=9.1.1.1/24                                 (4)
```

With this table, with no caching-aware packet classification, any TCP packet will always generate a megaflow that matches on TCP source and destination ports, because flow #3 matches on those fields. With priority sorting (Section 5.2), packets that match flow #2 can omit matching on TCP ports, because flow #3 is never considered. With staged lookup (Section 5.3), IP packets not destined to 9.1.1.1 never need to match on TCP ports, because flow #3 is identified as non-matching after considering only the IP destination address. Finally, address prefix tracking (Section 5.4) allows megaflows to ignore some of the bits in IP destination addresses even though flow #3 matches on the entire address.

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

Table 1: Performance testing results for classifier optimizations. Each row reports the measured number of Netperf `TCP_CRR` transactions per second, in thousands, along with the number of kernel flows, kernel masks, and user and kernel CPU usage.

Microflows	Optimizations	ktps	Tuples/pkt	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

Table 2: Effects of microflow cache. 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.

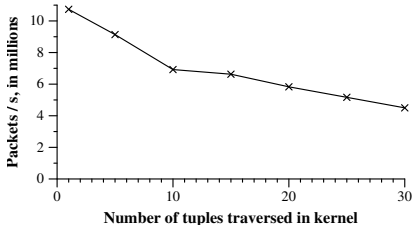


Figure 8: Forwarding rate in terms of the average number of megaflow tuples searched, with the microflow cache disabled.

Cache layer performance. We measured first the baseline performance of each Open vSwitch cache layer. In all following tests, Open vSwitch ran on a Linux server with two 8-core, 2.0 GHz Xeon processors and two Intel 10-Gb NICs. To generate many connections, we used Netperf's `TCP_CRR` test [25], which repeatedly establishes a TCP connection, sends and receives one byte of traffic, and disconnects. The results are reported in transactions per second (tps). Netperf only makes one connection attempt at a time, so we ran 400 Netperf sessions in parallel and reported the sum.

To measure the performance of packet processing in Open vSwitch userspace, we configured `ovs-vswitchd` to disable megaflow caching, by setting up only microflow entries in the datapath. As shown in Table 1, this yielded 37 ktps in the `TCP_CRR` test, with over one million kernel flow entries, and used about 1 core of CPU time.

To quantify the throughput of the megaflow cache by itself, we re-enabled megaflow caching, then disabled the kernel's microflow cache. Table 2 shows that disabling the microflow cache reduces `TCP_CRR` performance from 120 to 92 ktps when classifier optimizations are enabled. (When classifier optimizations are disabled, disabling the microflow cache has little effect because it is overshadowed by the increased number of trips to userspace.)

Figure 8 plots packet forwarding performance for long-lived flows as a function of the average number of tuples searched, with the kernel microflow cache disabled. In the same scenarios, with the microflow cache enabled, we measured packet forwarding performance of long-lived flows to be approximately 10.6 Mpps, independent of the number of tuples in the kernel classifier. Even searching only 5 tuples on average, the microflow cache improves performance by 1.5 Mpps, clearly demonstrating its value. To put these numbers in perspective in terms of raw hash lookup performance, we benchmarked our tuple space classifier in isolation: with a randomly generated table of half a million flow entries, the implementation is able to do roughly 6.8M hash lookups/s, on a single core – which translates to 680,000 classifications per second with 10 tuples.

Classifier optimization benefit. We measured the benefit of our classifier optimizations. Table 1 shows the improvement from individual optimizations and all of the optimizations together. Each optimization reduces the number of kernel flows needed to run the test. Each kernel flow corresponds to one trip between the kernel and userspace, so each reduction in flows also reduces userspace CPU time used. As can be seen from the table, as the number of kernel flows (Flows) declines, the number of tuples in the kernel flow table (Masks) increases, increasing the cost of kernel classification, but the measured reduction in kernel CPU time and increase in `TCP_CRR` shows that this is more than offset by the microflow cache and by fewer trips to userspace. The `TCP_CRR` test is highly sensitive to latency, demonstrating that latency decreases as well.

Comparison to in-kernel switch. We compared Open vSwitch to the Linux bridge, an Ethernet switch implemented entirely inside the Linux kernel. In the simplest configuration, the two switches achieved identical throughput (18.8 Gbps) and similar `TCP_CRR` connection rates (696 ktps for Open vSwitch, 688 for the Linux bridge), although Open vSwitch used more CPU (161% vs. 48%). However, when we added one flow to Open vSwitch to drop STP BPDU packets and a similar `iptables` rule to the Linux bridge, Open vSwitch performance and CPU usage remained constant whereas the Linux bridge connection rate dropped to 512 ktps and its CPU usage increased over 26-fold to 1,279%. This is because the built-in kernel functions have per-packet overhead, whereas Open vSwitch's overhead is generally fixed per-megaflow. We expect enabling other features, such as routing and a firewall, would similarly add CPU load.

8 Ongoing, Future, and Related Work

We now briefly discuss our current and planned efforts to improve Open vSwitch, and briefly cover related work.

8.1 Stateful Packet Processing

OpenFlow does not accommodate stateful packet operations, and thus, per-connection or per-packet forwarding state requires the controller to become involved. For this purpose, Open vSwitch allows running on-hypervisor "local controllers" in addition to a remote, primary controller. Because a local controller is an arbitrary program, it can maintain any amount of state across the packets that Open vSwitch sends it.

NVP includes, for example, a local controller that implements a stateful L3 daemon responsible for sending and processing ARPs. The L3 daemon populates an L3 ARP cache into a dedicated OpenFlow table (not managed by the primary controller) for quick forwarding of common case (packets with a known IP to MAC binding). The L3 daemon only receives packets resulting in an ARP cache miss and emits any necessary ARP requests to remote L3 daemons based on the packets received from Open vSwitch. While the connectivity between the local controller and Open vSwitch is local, the performance overhead is significant: a received packet traverses first from kernel to userspace daemon from which it traverses across a local socket (again via kernel) to a separate process.

For performance critical stateful packet operations, Open vSwitch relies on kernel networking facilities. For instance, a solid IP tunneling implementation requires (stateful) IP reassembly support. In a similar manner, transport connection tracking is a first practical requirement after basic L2/L3 networking; even most basic firewall security policies call for stateful filtering. OpenFlow is flexible enough to implement *static* ACLs but not stateful ones. For this, there's an ongoing effort to provide a new OpenFlow action that invokes a kernel module that provides metadata which the subsequent OpenFlow tables may use the connection state (new, established, related) in their forwarding decision. This "connection tracking" is the same technique used in many dedicated firewall appliances. Transitioning between kernel networking stack and kernel datapath module incurs overhead but avoids the duplication of functionality, critical in upstreaming kernel changes.

8.2 Userspace Networking

Improving the virtual switch performance through userspace networking is a timely topic due to NFV [9, 22]. In this model, packets are passed directly from the NIC to VM with minimal intervention by the hypervisor userspace/kernel, typically through shared memory between NIC, virtual switch, and VMs. To this end, there is an ongoing effort to add both DPDK [11] and netmap [30] support to Open vSwitch. Early tests indicate the Open vSwitch caching architecture in this context is similarly beneficial to kernel flow cache.

An alternative to DPDK that some in the Linux community are investigating is to reduce the overhead of going through the kernel. In particular, the SKB structure that stores packets in the Linux kernel is several cache lines large, contrary to the compact representation in DPDK and netmap. We expect the Linux community will make significant improvements in this regard.

8.3 Hardware Offloading

Over time, NICs have added hardware offloads for commonly needed functions that use excessive host CPU time. Some of these features, such as TCP checksum and segmentation offload, have proven very effective over time. Open vSwitch takes advantage of these offloads, and most others, which are just as relevant to virtualized environments. Specialized hardware offloads for virtualized environments have proven more elusive, though.

Offloading virtual switching entirely to hardware is a recurring theme (see, *e.g.,* [12]). This yields high performance, but at the cost of flexibility: a simple fixed function hardware switch effectively replaces the software virtual switch with no ability for the hypervisor to extend its functionality. The offload approach we currently find most promising is to enable NICs to accelerate kernel flow classification. The Flow Director feature on some Intel NICs has already been shown to be useful for classifying packets to separate queues [36]. Enhancing this feature simply to report the matching rule, instead of selecting the queue, would make it useful as such for megaflow classification. Even if the TCAM size were limited, or if the TCAM did not support all the fields that the datapath uses, it could speed up software classification by reducing the number of hash table searches – without limiting the flexibility since the actions would still take place in the host CPU.

8.4 Related Work

Flow caching. The benefits of flow caching generally have been argued by many in the community [4, 13, 17, 31, 41]. Lee et al. [21] describes how to augment the limited capacity of a hardware switch's flow table using a software flow cache, but does not mention problems with flows of different forms or priorities. CacheFlow [13], like Open vSwitch, caches a set of OpenFlow flows in a fast path, but CacheFlow requires the fast path to directly implement all the OpenFlow actions and requires building a full flow dependency graph in advance.

Packet classification. Classification is a well-studied problem [37]. Many classification algorithms only work with static sets of flows, or have expensive incremental update procedures, making them unsuitable for dynamic OpenFlow flow tables [7, 8, 33, 38, 40]. Some classifiers require memory that is quadratic or exponential in the number of flows [8, 20, 35]. Other classifiers work only with 2 to 5 fields [35], whereas OpenFlow 1.0 has 12 fields and later versions have more. (The effective number of fields is much higher with classifiers that must treat each bit of a bitwise matchable field as an individual field.)

9 Conclusion

We described the design and implementation of Open vSwitch, an open source, multi-platform OpenFlow virtual switch. Open vSwitch has simple origins but its performance has been gradually optimized to match the requirements of multi-tenant datacenter workloads, which has necessitated a more complex design. Given its operating environment, we anticipate no change of course but expect its design only to become more distinct from traditional network appliances over time.

References

[1] T. J. Bittman, G. J. Weiss, M. A. Margevicius, and P. Dawson. Magic Quadrant for x86 Server Virtualization Infrastructure. Gartner, June 2013.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proc. of SIGCOMM*, 2007.

[4] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In *Proc. of HotNets*, 2008.

[5] Crehan Research Inc. and VMware Estimate, Mar. 2013.

[6] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.

[7] A. Feldman and S. Muthukrishnan. Tradeoffs for Packet classification. In *Proc. of INFOCOM*, volume 3, pages 1193–1202 vol.3, Mar 2000.

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

[9] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of NSDI*, Apr. 2014.

[10] IEEE Standard 802.1ag-2007: Virtual Bridged Local Area Networks, Amendment 5: Connectivity Fault Management, 2007.

[11] Intel. *Intel Data Plane Development Kit (Intel DPDK): Programmer's Guide*, October 2013.

[12] Intel LAN Access Division. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. `http://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf`, January 2011.

[13] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite CacheFlow in Software-Defined Networks. In *Proc. of HotSDN*, 2014.

[14] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880 (Proposed Standard), June 2010.

[15] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Proc. of NSDI*, 2013.

[16] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. of NSDI*, 2012.

[17] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting Route Caching: The World Should Be Flat. In *Proc. of PAM*, 2009.

[18] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster. SAX-PAC (Scalable And eXpressive PAcket Classification). In *Proc. of SIGCOMM*, 2014.

[19] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proc. of NSDI*, Seattle, WA, Apr. 2014.

[20] T. Lakshman and D. Stiliadis. High-speed Policy-based Backet Forwarding Using Efficient Multi-dimensional Range Matching. *SIGCOMM CCR*, 28(4):203–214, 1998.

[21] B.-S. Lee, R. Kanagavelu, and K. M. M. Aung. An Efficient Flow Cache Algorithm with Improved Fairness in Software-Defined Data Center Networks. In *Proc. of Cloudnet*, pages 18–24, 2013.

[22] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. of NSDI*, Apr. 2014.

[23] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[24] Microsoft. Hyper-V Virtual Switch Overview. `http://technet.microsoft.com/en-us/library/hh831823.aspx`, September 2013.

[25] The Netperf homepage. `http://www.netperf.org/`, January 2014.

[26] Open vSwitch – An Open Virtual Switch. `http://www.openvswitch.org`, September 2014.

[27] OpenFlow. `http://www.opennetworking.org/sdn-resources/onf-specifications/openflow`, January 2014.

[28] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047 (Informational), Dec. 2013.

[29] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *Proc. of HotNets*, Oct. 2009.

[30] L. Rizzo. netmap: A novel framework for fast packet I/O. In *Proc. of USENIX Annual Technical Conference*, pages 101–112, 2012.

[31] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang. Leveraging Zipf's Law for Traffic Offloading. *SIGCOMM CCR*, 42(1), Jan. 2012.

[32] N. Shelly, E. Jackson, T. Koponen, N. McKeown, and J. Rajahalme. Flow Caching for High Entropy Packet Fields. In *Proc. of HotSDN*, 2014.

[33] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *Proc. of SIGCOMM*, 2003.

[34] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *Proc. of SIGCOMM*, 1999.

[35] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer Four Switching. In *Proc. of SIGCOMM*, 1998.

[36] V. Tanyingyong, M. Hidell, and P. Sjodin. Using Hardware Classification to Improve PC-based OpenFlow Switching. In *Proc. of High Performance Switching and Routing (HPSR)*, pages 215–221. IEEE, 2011.

[37] D. E. Taylor. Survey and Taxonomy of Packet Classification Techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.

[38] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *Proc. of SIGCOMM*, Aug. 2010.

[39] VMware. vSphere Distributed Switch. `http://www.vmware.com/products/vsphere/features/distributed-switch`, September 2014.

[40] T. Y. C. Woo. A Modular Approach to Packet Classification: Algorithms and Results. In *Proc. of INFOCOM*, volume 3, pages 1213–1222 vol.3, Mar 2000.

[41] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. In *Proc. of SIGCOMM*, 2010.

[42] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance Ethernet forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 97–108, New York, NY, USA, 2013. ACM.