SoftFlow: A Middlebox Architecture for Open vSwitch

Ethan J. Jackson, *University of California, Berkeley;* **Melvin Walls,** *Penn State Harrisburg and University of California, Berkeley;* **Aurojit Panda,** *University of California, Berkeley;* **Justin Pettit, Ben Pfaff, and Jarno Rajahalme,** *VMware, Inc.;* **Teemu Koponen,** *Styra, Inc.;* **Scott Shenker,** *University of California, Berkeley, and International Computer Science Institute*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/jackson

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16) is sponsored by USENIX.

SoftFlow: A Middlebox Architecture for Open vSwitch

Ethan J. Jackson[†] Melvin Walls[¶†] Aurojit Panda[†] Justin Pettit[*]
Ben Pfaff[*] Jarno Rajahalme[*] Teemu Koponen[‡] Scott Shenker[†$]
[*]VMware, Inc. [†]UC Berkeley [‡]Styra, Inc. [$]ICSI [¶]Penn State Harrisburg

Abstract

Open vSwitch is a high-performance multi-layer virtual switch that serves as a flexible foundation for building virtualized, stateless Layer 2 and 3 network services in multi-tenant datacenters. As workloads become more sophisticated, providing tenants with virtualized middlebox services is an increasingly important and recurring theme, yet it remains difficult to integrate these stateful services efficiently into Open vSwitch and its OpenFlow forwarding model: middleboxes perform complex operations that depend on internal state and inspection of packet payloads – functionality which is impossible to express in OpenFlow. In this paper, we present SoftFlow, an extension of Open vSwitch that seamlessly integrates middlebox functionality while maintaining the familiar OpenFlow forwarding model and performing significantly better than alternative techniques for middlebox integration.

1 Introduction

With the rise of network virtualization, the primary provider of network services in virtualized clouds has migrated from the physical datacenter fabric to the hypervisor virtual switch. This trend demands virtual switches implement *virtual networks* that faithfully reproduce complex L2—L3 network topologies that were once entirely the concern of network hardware.

As network virtualization systems mature and workloads increase in sophistication and complexity, pressure continues to mount on virtual switches to provide more advanced features without sacrificing flexibility or performance. In particular, middleboxes – firewalls, NATs, load balancers, and the like – that are ubiquitous in enterprise networks [32] have begun to make their way into network virtualization systems.

Open vSwitch (OVS) – the open source virtual switch utilized by a majority of these systems – is not immune to this pressure. Its flow based forwarding model (based on OpenFlow) makes it particularly well suited to stateless L2—L3 forwarding, allowing it to achieve a high level of generality without sacrificing performance [23]. However, extending this model to middleboxes has proven difficult due to three fundamental challenges:

- Open vSwitch (and OpenFlow) models packet processing as a series of flow tables operating over packet headers. Middleboxes, on the other hand, rely on per-connection state and inspection of packet payloads that are hard to express in this model.
- In order to achieve reasonable performance, Open vSwitch uses a flow caching algorithm that depends necessarily on the stateless nature of OpenFlow to produce consistent results – packets with the exact same header must be forwarded the exact same way every single time. Middleboxes' reliance on internal state and inspection of packet payloads causes them to make *different* forwarding decisions for packets with the same header. This breaks the fundamental assumptions of the flow cache.
- Packet parsing and classification are elementary operations among all network services that long complex service chains must perform many times for a given packet. While it is feasible to integrate middleboxes with Open vSwitch using virtual machines, it's unclear how to share this work across middleboxes as Open vSwitch is able to for stateless L2—L3 OpenFlow pipelines.

In this paper we design SoftFlow, a data plane forwarding model with unified semantics for all types of packet operations. SoftFlow is an extension of Open vSwitch designed around three design principles:

Maintain the Open vSwitch forwarding model. Open vSwitch is built on OpenFlow, which has arguably helped it achieve the wide deployment it enjoys today and we see no reason to abandon it. A great deal of traditional middlebox functionality, *e.g.,* L2, L3, and ACL processing, can be implemented naturally as flows, leaving us with only a small subset of functionality that needs special processing: operations which require per-packet state and operations which inspect packet payloads. These operations can be handled with our SoftFlow extensions.

Reduce packet classifications. On general-purpose processors algorithmic packet classification is expensive. In our experience, it frequently consumes the majority of datapath CPU time and experiments in §7 indicate the same.[1] We aim to extend the benefits of Open vSwitch flow caching to middleboxes by designing for *middlebox aware* flow caching to exploit localities *across packets*, and shared packet classification *between middlebox services* to mitigate redundant computation. In doing so, we reduce the overhead of classification-heavy middleboxes like NATs and firewalls.

Increase processing locality. Running services outside of the virtual switch on separate VMs or processes provides strong isolation, both in terms of performance and memory. However, this isolation comes at a cost – performance suffers

[1]There is a vast literature on the subject of algorithmic packet classification [5, 14, 33–35]. However, we note that any of these complex algorithms performed *repeatedly* per-packet is likely to dominate processing time.

due to high virtual I/O overhead and CPU cache misses as packets traverse multiple cores. While isolation is generally assumed necessary in the NFV literature [7, 19], in systems where service implementations can be carefully vetted by a single vendor, we believe it is less critical. We choose to sacrifice isolation so we may adopt a run-to-completion model in which packets are processed by a single core, from reception, through various services, and finally to transmission. This choice leads to a factor of 2 performance boost over an NFV-style VM-based implementation.

We have implemented a prototype of SoftFlow on top of the Open vSwitch port to DPDK [9] that we are currently evaluating for inclusion in upstream Open vSwitch. In addition to the SoftFlow extensions to Open vSwitch, we have also implemented a number of service pipelines to validate the generality of the design. In what follows:

- We provide the design and implementation of an extension to Open vSwitch that supports generic L4–L7 middleboxes.
- We show how to integrate L4–L7 services with flow caching and a run-to-completion forwarding model, and provide evidence in §7 that the performance benefits of an integrated approach are as much as a factor of 2 better than VM-based middleboxes.
- We provide a design for integrating NIC hardware packet classification offload with SoftFlow, without restricting the generality of software packet processing. We show in §7 this optimization could improve forwarding rates a further 5%–90% for realistic workloads.

2 Background

While SoftFlow has broad applications, we designed it specifically to solve challenges present in the network virtualization systems [12, 16, 37]. These systems provide virtual networking to a cloud of virtual machines (or containers) running on thousands of hypervisors. Each hypervisor runs a sophisticated virtual switch, typically OVS, which is used to form an overlay of densely interconnected tunnels. In addition to the hypervisors, each system has a *gateway* which sits in between virtual networks and legacy physical networks hosting non-virtualized workloads. Gateways are typically implemented as a dedicated OVS instance running on a commodity server.

These systems have proven successful for basic stateless L2/L3 networking. However, as deployments mature and more sophisticated workloads are migrated onto virtual networks, the demand for more advanced service typical of the ubiquitous middleboxes in enterprise networks emerges. Firewalls, NATs, load-balancers, and the like are essential components of modern networks, which administrators expect to be available in their virtualized deployments.

Our primary motivation for developing SoftFlow is to support these complex middlebox services in network virtualization platforms. The system must perform well both on hypervisors, where efficiency is paramount to maximize resources available to VMs, and on gateways where performance both in terms of throughput and ability to scale to thousands of tenants, is highly valued. The peculiarities of this environment have led us to a somewhat different emphasis than is typical in the NFV literature.

- The environment is multi-tenant, but not necessarily multi-vendor. We expect most middlebox functionality to be developed specifically for Open vSwitch and carefully vetted by the community. For these reasons, strong isolation between services, while nice to have, is not critical.
- Hypervisors must balance performance and efficiency. Cloud operators are typically willing to devote a core or two to networking, but taking up the majority of hypervisor CPU resources for networking is unacceptable.
- Network virtualization systems already rely heavily on Open vSwitch and forcing them to migrate wholesale to a foreign switch based on service chains of black-box virtual machines would be, at best, burdensome.

In searching for a solution, we evaluated, attempted, and ultimately rejected two common approaches to middlebox development seen today. In the rest of the section, we discuss both in turn: the black-box model which hides middlebox complexity in virtual machines, and the "pure SDN" model which attempts to build middleboxes on top of OpenFlow.

2.1 Black Boxes

Perhaps the most common approach to building virtualized middleboxes in the literature is what we call the *black box* model. In this approach, each middlebox is a fully isolated virtual machine (or container) with a series of virtual ports connecting it to the hypervisor virtual switch. The appeal of this approach is obvious: it provides a straightforward migration path to the cloud for middlebox vendors. However, it comes with costs:

- Middlebox implementations tend to be developed and managed completely independently of the rest of the virtual network. This significantly complicates the control plane design, which must manage (at least) two completely separate systems.
- Packet transmission between cores (necessary to shuffle packets between VMs and the hypervisor) is costly. Modern systems [7, 17, 19, 23, 24, 28] mitigate this cost through shared memory, sometimes even employing zero-copy techniques. However, as we show in §7 for realistic workloads, the benefits of zero-copy I/O pale in comparison to the performance of a run-to-completion forwarding model.
- Virtual machine instantiation and management has significant overhead. In the systems that SoftFlow targets, gateways often need to run per-tenant middleboxes. For large clouds, this can add up to hundreds or thousands of instances per gateway. For instance, the Network Virtualization Platform (NVP) [16], a commercial software defined network, used Linux network namespaces (which

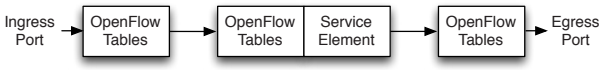


Figure 1: A conceptual model of the SoftFlow datapath. SoftFlow actions offload their classification to OpenFlow tables.

are significantly lighter than VMs) as its gateway middlebox implementation, but yet quickly encountered scale issues as management of such heavyweight appliances proved challenging. In SoftFlow, however, each middlebox is a simple function pointer in the Open vSwitch process address space – thousands can coexist trivially.

2.2 Pure OpenFlow Middleboxes

To a very limited extent, trivial middleboxes can be built on top of OpenFlow. In our quest to support middleboxes for NVP [22] we did just this with limited success. We briefly survey the pure OpenFlow middleboxes techniques we attempted, and why they fail to generalize beyond all but the simplest use cases.

When one attempts to build a pure OpenFlow middlebox, one must deal with the limitations of OpenFlow: its inability to handle stateful operations and deep packet inspection. The first attempted solution often involves a *reactive* OpenFlow model, where packets needing middlebox processing are punted to the controller for handling. It is not difficult to achieve functionally "correct" behavior with this strategy, but it is challenging to scale the approach to large networks with thousands of middleboxes forwarding millions of packets per second each.

The reactive model can be optimized by adopting a local controller co-located with each switch to handle certain types of packets. This approach scales well for services which only need to inspect the occasional packet but, beyond that, sending each packet through the Open vSwitch slow path to the local controller is both more complex and expensive than the black-box model.

Given the relatively high overhead of a local controller, a further optimization employed by Open vSwitch allows the switch *itself* to modify its own flow tables through the *learn action*. While originally designed to do L2 learning, one can implement primitive stateful services with it. For instance, a simple stateful firewall can be built by "learning a hole" (installing a new flow) for the reverse traffic of each new connection. In fact, this approach has been adopted and deployed in NVP as a primitive virtual firewall. However, it can't do true connection tracking which requires verifying the TCP state machine, implement middleboxes which require deep packet inspect, or even those that require fast changing per-packet state.

SoftFlow models the datapath as a stateless processing pipeline, implemented in OpenFlow, which forwards packets through stateful processing modules we call SoftFlow actions. These modules perform the complex processing which

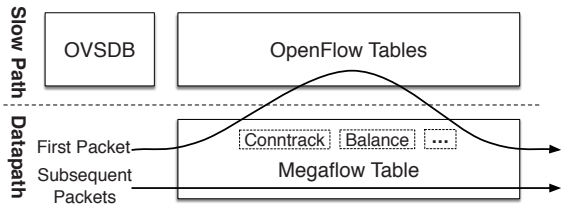


Figure 2: The components of Open vSwitch. The datapath maintains a megaflow table and the implementations of all SoftFlow actions. Packets not found in the cache are punted to the OpenFlow tables above.

OpenFlow can't support and, when finished, send packets to the next OpenFlow table for further processing. This process repeats as many times as necessary, allowing multiple stateless and stateful stages to operate on each packet.

Unlike virtual machines operating under the black-box model, SoftFlow actions can offload internal packet classifications to OpenFlow tables. For example, a controller may choose to build a firewall from an ACL table implemented in OpenFlow and a SoftFlow action which implements connection tracking. This ability to offload packet classifications has two key design benefits: First, it simplifies middleboxes by relieving them of the need to implement their own packet classification algorithms. Second, it allows middlebox packet classifications to participate in OVS flow caching along with OpenFlow tables. As a result, multiple steps from both OpenFlow tables and SoftFlow actions can be combined into a single classification per packet in the flow cache. This greatly reduces redundant packet classifications in the datapath. Figure 1 depicts this model.

3 SoftFlow Design

3.1 Flow Caching and Services

SoftFlow builds on top of Open vSwitch and inherits its flow caching architecture [25, 31] that we briefly review here. The architecture is split into two major components, a slow path and a datapath, as illustrated in Figure 2. The datapath is responsible for per-packet forwarding, which it achieves using a *megaflow* table: a giant flat (priority-less) classifier which supports bit-wise wildcarding.[2] Each rule in the megaflow table carries with it a list of actions (*e.g.,* modify headers, forward) which are performed on each packet that matches. Packets for which the datapath has no matching megaflow are punted to the slow path where the OpenFlow implementation resides. These "missed" packets are forwarded through a series of OpenFlow tables that generate a wildcard mask and action list making up a new megaflow entry. This new megaflow entry is then installed in the datapath, so that future packets can avoid the expense of a slow path traversal.

[2]Below the megaflow table there is also an exact match cache in the OVS DPDK implementation. Like Open vSwitch, SoftFlow takes advantage of this cache, but we don't describe it in detail here for brevity.

The Open vSwitch cache hierarchy described above was designed specifically to suit the needs of OpenFlow, not to support complex stateful processing necessary to implement middleboxes. In fact, as we developed SoftFlow, we identified two key limitations which must be resolved to support these workloads:

- The flow cache hierarchy is built on the assumption that each packet (as defined by its header) will be processed exactly the same way by a given set of OpenFlow tables every time. This is why a long series of OpenFlow tables can be compressed to a single packet classification in the megaflow table. SoftFlow actions, on the other hand, can make forwarding decisions based on *internal state* and *packet payload* that, from the perspective of the flow cache, are completely *nondeterministic*. This means that the flow cache must be prepared to re-classify each packet after every SoftFlow action invocation.
- The Open vSwitch slow path must know if an OpenFlow action will modify a packet, so it can take this modification into account when executing future OpenFlow lookups. While quite practical for simple OpenFlow actions, for complex SoftFlow actions this would require an implementation of the action to live in both the slow path and datapath. Furthermore, to behave correctly these two implementations would have to synchronize their internal state. While theoretically feasible, this would add a great deal of avoidable complexity and overhead.

We now consider the implementation of SoftFlow on top of Open vSwitch with these two challenges in mind. Our solutions to these problems flow naturally from three key design decisions:

Datapath Exclusive Actions. SoftFlow actions are implemented entirely in the datapath, and as a result, the slow path has no semantic understanding of SoftFlow action behavior. A pipeline of OpenFlow tables cannot continue after a SoftFlow action has executed as the slow the path doesn't know how to update the packet for the next OpenFlow table. Instead, SoftFlow actions *consume* the packets they operate on just as a VM does.

SoftFlow Stages. In order to continue a forwarding pipeline after SoftFlow actions execute, we require the controller to explicitly handle packets emerging from them at the beginning of the OpenFlow pipeline just as if they had emerged from a VM. Semantically, this requires a megaflow lookup after each action invocation (which we will optimize away shortly).

To achieve this we introduce the concept of a SoftFlow stage. Each packet entering the system is tagged with a stage id, `sf_stage`, initially set to 0. Immediately after a SoftFlow action executes, `sf_stage` increments and the packet is re-injected at the bottom of the cache hierarchy. Crucially, `sf_stage` can be matched at all layers of the cache hierarchy including the slow path, making it easy for the controller to distinguish packets emerging from SoftFlow actions from those emerging from VMs or NICs.

Metadata	Description
`sf_metadata`	Metadata register accessible by SoftFlow actions.
`sf_action`	Name of the last executed SoftFlow action.
`sf_stage`	Stage incremented after each action invocation.
`sf_coalesce`	Boolean signalling whether it is ok to coalesce.
`sf_argument`	Runtime configuration argument set by controller.

Figure 3: SoftFlow relies on a number of metadata values embedded in each packet structure. These values, and what they're for, are briefly summarized in this table..

Stage Coalescing. The Open vSwitch flow cache requires that all packets with a particular header be forwarded the exact same way every single time. This invariant holds easily for OpenFlow, but for SoftFlow actions, which can make header modifications based on internal state and packet payloads, two packets with the same header may be treated very differently. The naïve solution to this problem requires a full megaflow lookup after each SoftFlow action invocation, however, this is often unnecessary: many common middleboxes do not modify packet headers at all (*e.g.,* firewall, IDS, IPS), or if they do, modifications are deterministic given the input packet headers. On the other hand, some actions may make packet modifications on a per packet basis and really do require a megaflow lookup after each invocation.

If a SoftFlow action does not make any modifications that would require an additional megaflow lookup for that packet header, it can take advantage of a novel optimization called *stage coalescing* to avoid it. After executing, the action signals whether a new lookup is necessary by writing a boolean into the `sf_coalesce` packet metadata. If `true`, SoftFlow can skip the next megaflow lookup, instead, simply executing the actions that follow the current SoftFlow action.

The optimization is set up at megaflow installation time. If a new megaflow ends in a SoftFlow action that supports coalescing, SoftFlow tracks the packet as it flows through the stages ahead of it and *appends* the actions executed in those stages to the original megaflow's action list. Thus, when future packets hit this megaflow, they have access to all of the actions they'll need without having to execute additional megaflow lookups. Furthermore, if at any time a particular SoftFlow action needs to, it can always set `sf_coalesce` to `false` forcing a new megaflow lookup after its execution.

3.2 SoftFlow and OpenFlow

We expect a controller to configure and instantiate SoftFlow actions out of band. In our prototype implementation the set of available actions and their configuration is hard coded, but it would be simple to dynamically load new actions at runtime and configure them through OVSDB.

We used OpenFlow's vendor extension mechanism, to

add a new OpenFlow action, `softflow`, in which the controller embeds the name of the SoftFlow action to be executed, and an integer `sf_argument`.[3] This argument, used as a runtime configuration parameter, is passed to the SoftFlow action on each invocation.

Additionally, we augment OpenFlow by adding a per packet register, `sf_metadata`, which SoftFlow actions can use to exchange information. On invocation, the current value of the register is passed to the action which may be read or written it at will. After invocation future SoftFlow actions or OpenFlow tables can match on the new value and alter their behavior accordingly. In addition to the metadata register, the OpenFlow tables can also match on the `sf_stage`, and `sf_action`, the name of the most recently executed SoftFlow action.

3.3 Limitations

The tight integration of SoftFlow with the Open vSwitch data plane is not without its limitations which we discuss briefly below.

SoftFlow makes no attempt to isolate actions from each other or the rest of the platform. Since all actions share the same process address space, this implies that buggy or malicious actions can crash the switch or, worse, read from or write to other action's memory. Additionally, even if all actions are trusted and well implemented, SoftFlow provides no mechanism to fairly allocate CPU or memory resources to actions. Thus, SoftFlow is only suitable for carefully vetted trusted actions. In effect, SoftFlow stakes out an extreme position in the trade-off between strong isolation (and thus the performance cost represented by process/VM isolation) and a faster run-to-completion forwarding model.

SoftFlow is not well suited for middleboxes that rely heavily on buffering packets – specifically systems that participate in TCP connections like HTTP proxies, certain types of Intrusion Detection Systems, and some WAN optimizers. Such middleboxes require a SoftFlow action to execute the majority of processing in separate background threads, in effect preventing the run-to-completion forwarding model. While this can be implemented, after all SoftFlow actions do run arbitrary code, it would be simpler to use a virtual machine or process instead.

Finally, we note that SoftFlow makes no particular provisions for fault tolerance. In the event of failure, fail-over to a backup switch must be handled by an out of band mechanism, as is commonly done in OVS appliances today. In SoftFlow this problem is complicated by the fact that middleboxes often have internal state that must be migrated to the backup copy for correct operation. SoftFlow leaves this problem to the action implementation and makes no attempt to solve it in the framework.

[3] In future, we expect to support multiple arguments and, perhaps, more complex data types like strings. For now, however, the single argument has proven sufficient.

4 Service Design

In this section we describe the design of two software forwarding pipelines built on SoftFlow: a simple stateful firewall and a load-balancer. Later we evaluate these pipelines in §7.

4.1 Stateful Firewall

Firewalls are built of two primary components: a packet classifier that implements an Access Control List (ACL), and a connection tracker that keeps track of transport connection state. While ACLs are simple to express as OpenFlow tables, the connection tracker is difficult to express in standard OpenFlow. It's designed to allow all packets from established connections, while only allowing new connections which satisfy the ACL. Depending on the sophistication of the firewall, connection tracking may imply not only tracking TCP connection state, but also validating TCP sequence numbers.

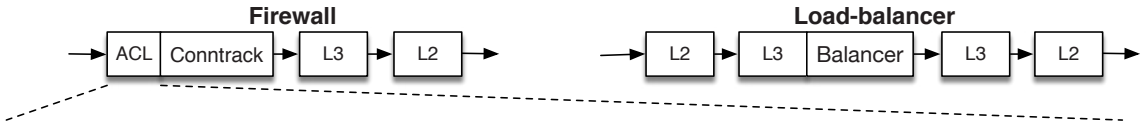
Our firewall implements its ACL table in OpenFlow, which works cooperatively with a SoftFlow `conntrack` action responsible for connection tracking. In our prototype implementation `conntrack`'s internals are based on a connection tracker designed for OVS-DPDK [2] that is itself based on the BSD firewall.

Our firewall pipeline begins when packets are forwarded to the ACL OpenFlow table. If any ACLs successfully match a packet, it passes to the `conntrack` action with `sf_argument` 1. This `sf_argument` value indicates to the action that the packet successfully matched an "allow" ACL and, if necessary, a new connection entry may be created. The table also has a low-priority default rule for those packets that don't match any ACL. These packets are still passed to the SoftFlow action, but with `sf_argument` value 0, indicating that the packets failed to match the ACL table, and should only be forwarded if they belong to an existing connection.

The `conntrack` action executes and, in the process, writes to the packet's `sf_metadata` the value `1` if the packet is allowed, or `0` if it should be dropped. When finished, control returns back to the datapath which increments `sf_stage` and passes the packet back into the OpenFlow tables.

To avoid conflicts with other rules, the OpenFlow table matches on `sf_stage=1` and `sf_action=conntrack` to catch packets received from the connection tracker. These rules either drop failures or instruct successes to proceed to the next table. Finally, the packet traverses L3 and L2 tables which determine the appropriate output port. Figure 4 depicts the resulting SoftFlow firewall pipeline as well as its first flow table, both responsible for processing packets incoming from the NIC and from the conntrack service.

Packet walkthrough. A packet entering the datapath is first checked against the megaflow cache. If empty, the packet enters the OpenFlow slow path, which is responsible for compiling a datapath action list and a wildcard mask for a new megaflow cache entry. The packet begins in the initial




```
priority=100 sf_stage=1 sf_action=conntrack sf_metadata=1 -> goto L3 table
priority=100 sf_stage=1 sf_action=conntrack sf_metadata=0 -> drop
priority=100 sf_stage=0 ip_dst=10.0.0.3 tcp_src=80        -> softflow:conntrack:1
priority=0   sf_stage=0                                    -> softflow:conntrack:0
```

Figure 4: A simple pipeline for a firewall (left) and load-balancer (right). Below, a simple firewall ACL table that allows 10.0.0.3 to initiate TCP connections on port 80 and allows all return traffic; all other traffic is dropped. The first two flow entries match traffic coming back from the conntrack service. The contents of the firewall L2 and L3 forwarding tables, as well as LB forwarding tables, are omitted for brevity.

OpenFlow table, a packet classification is performed, and a matching ACL is chosen.

The OpenFlow action corresponding to the matching ACL indicates that the conntrack service should be invoked. However, the slow path *does not execute* the service itself as the actual implementation lives entirely in datapath. Instead, the slow path simply appends the SoftFlow action, with some additional metadata, to the action list gathered so far and passes this list to the datapath as a new megaflow cache entry.

The datapath then executes the action list. The final item in the list is a reference to the SoftFlow action along with a function pointer and the `sf_argument` necessary to invoke it. The datapath executes the action by making a simple function call, passing along the packet and `sf_argument`.

The action executes, runs the packet through its connection tracking logic, and sets `sf_metadata` to `1` meaning "pass". Since "pass" is the typical result, the action also sets `true` to `sf_coalesce` meaning this packet may be coalesced. Control returns to the SoftFlow datapath which increments `sf_stage`, and sets `sf_action` to "conntrack".

At this point, a normal Open vSwitch would install the rule in its megaflow cache. However, since `sf_coalesce` is `true`, coalescing is enabled for this packet. Therefore it is passed back to the slow path once again.

Receiving the packet for the second time, the slow path again runs the packet through the standard OpenFlow pipeline starting at the first table. Since `sf_stage` is `1`, `sf_action` is "conntrack", and `sf_metadata` is `1`, the first rule in the figure matches and the packet proceeds to the L2/L3 tables.

Again, the slow path assembles a mask and action list, and hands them to the datapath. The cache executes the action list just as before, then *appends* the results to the action list collected from the previous slow path invocation. By doing so, the flow cache coalesces what would have been two stages into a single megaflow, effectively reducing the number of megaflow lookups for future packets to one.

Since no more SoftFlow actions are executed, the process terminates and the cache entry is finally installed. Future packets entering the system perform a megaflow lookup to find this newly installed megaflow and execute the action list without the overhead of a miss or several packet classifications. In the common case, this megaflow lookup will be handled by the exact-match cache, requiring just a single hash table lookup in addition to the cost of connection tracking.

4.2 Load Balancer

We now detail a load balancer (evaluated in §7) which balances TCP connections among a configurable number of backends. The load balancer is constructed from two pieces: an abstract `balancer` SoftFlow action and a controller-provided OpenFlow table which acts on the action's decisions. The load balancer assigns each connection to one of eight buckets and writes its choice to `sf_metadata`.

Just like the firewall, after the `balancer` action is executed, the packet is passed back to the OpenFlow table. In this table, a set of rules is installed that map each `sf_metadata` value to a backend host. The OpenFlow table transforms the packet accordingly (by setting the appropriate destination MAC and IP) before forwarding it to the next table for an L3 lookup, L2 lookup, and eventual transmission.

Packet walkthrough. Similar to the firewall, packets entering the datapath traverse the cache hierarchy eventually ending up in the slow path. The slow path compiles a new megaflow for installation that references the `balancer`. The datapath executes the action list, including `balancer` which takes the following steps:

- Assuming the packet belongs to a new connection, the `balancer` implementation creates a connection identifier for the packet, and assigns it to the least loaded of eight[4] available buckets.
- A map is updated noting which bucket this connection is assigned to so that future packets are forwarded consistently.
- The bucket's byte count is incremented with the packet's size, so that future packets can make appropriate balancing decisions.
- The packet's `sf_metadata` register is updated with the

[4]Note that eight was chosen arbitrarily, any small number would do.

ID of the chosen bucket.

- Finally, the balancer returns `false` to the SoftFlow datapath indicating that the result cannot be coalesced.

From the perspective of OpenFlow, the load balancer's behavior is *nondeterministic* regarding L2–L4 input headers. Its choice of bucket is based on internally maintained load information that OpenFlow does not have access to. Therefore, `balancer` always returns `false` to the datapath indicating that coalescing should never occur. For our packet, this means the megaflow is installed immediately, just as it would have been with standard Open vSwitch.

After the megaflow is installed, the packet is re-injected at the bottom of the cache hierarchy where it eventually finds its way back to the slow path OpenFlow tables. The packet traverses the OpenFlow tables, various transformations are made, and again a new megaflow is created. This time the megaflow is scoped to `sf_stage=1`, so that future packets emerging from the `balancer` action that are assigned to the same bucket, can skip the slow path entirely and need only perform a packet classification in the datapath. Once enough traffic has been processed to set up a megaflow per bucket, future packets can be forwarded without involving the slow path at all, but at the cost of two classifications: one to decide that the packet requires load-balancing and one to decide what actions to take based on the load-balancer's decision.

5 Implementation

In this section, we discuss details of our prototype SoftFlow implementation built on the Open vSwitch port to DPDK, OVS DPDK. In our prototype, a configurable number of cores are dedicated to forwarding, each of which runs a dedicated polling thread. Each NIC has a dedicated receive queue per thread over which packets are load balanced using a simple hash over the 5-tuple. Polling threads process packets using a run-to-completion model, meaning that the core which receives a packet sees it through all the way to transmission, never handing it to another thread for processing. Our prototype implementation is built on top of OVS 2.4 with a series of patches totalling approximately 1800 lines of code not including SoftFlow action implementations.

5.1 Service API

In our prototype, service implementations are directly compiled and linked to the Open vSwitch process, and statically configured for simplicity. This prototype mechanism could be replaced with a dynamic library loading of the services through a stable ABI with configuration through OVSDB.

SoftFlow maintains a global list of all SoftFlow services which, today, is simply hard-coded in the source (though future versions will allow it to be dynamically loaded at runtime). As Open vSwitch boots, it initializes each service using their internal initialization functions, and makes the services available to OpenFlow and the datapath for use.

In SoftFlow, services can form groups for their internal state sharing purposes. Thus, the actual service registration takes place at the group level: services are constructed as groups of actions and the service group is registered through a `sf_group` structure. In it, references to the service element specific initialization structures are provided.

Each service element provides a `sf_service` structure which specifies an instance initialization function (provided with a reference to the group) as well as a function for the SoftFlow datapath to invoke to process packets:

```
struct sf_service {
  const char *name;

  /* Construct and return a new instance. */
  void *(*init)(struct sf_group *group);

  /* Process a batch of packets. */
  void (*ingress)(struct pkt **pkts, size_t n,
                  void *instance);
};
```

This is the *entire* interface an action has to implement. Afterward, OpenFlow entries can refer to the action and the SoftFlow datapath can invoke the ingress function to execute batches of packets. The function's arguments are a batch of packets and an "instance" pointer to service internal state. Packet metadata is held within the packet construct.

5.2 Datapath Integration

While at the OpenFlow protocol level SoftFlow actions are quite similar to standard OpenFlow actions, their actual implementation is completely different from these simple procedures.

At OpenFlow rule install time, the SoftFlow action is translated to an internal representation which directly holds a pointer to the `ingress()` function and the metadata necessary to call it. When the slow path encounters this action, it in turn converts the packet to a flow cache specific internal representation which then contains the necessary information to invoke the action; that is, ingress function, instance pointer, and `sf_argument`. Thus, the rest of the flow cache can remain unaware of the intricacies of SoftFlow action invocation, and instead simply call a function pointer.

Coalescing. When a megaflow cache miss occurs in the standard Open vSwitch implementation, the packet is passed to the slow path, a new megaflow entry is returned, the entry is installed, and the packet is forwarded based on this new entry. In SoftFlow, however, this simple process must be enhanced to support stage coalescing. After receiving a new megaflow from the slow path, if the last action is a SoftFlow service, a different procedure is executed:

- The action list associated with the new megaflow is executed, including the final SoftFlow action.
- The SoftFlow action updates the `sf_metadata` and (let's assume for this example) sets `sf_coalesce` to `true`.

- The `sf_stage` increments, and `sf_action` updates.
- The packet is passed back to the slow path which generates a new megaflow.
- The new megaflow is merged with the previous megaflow, and its actions are appended to the currently accumulating list.
- The process repeats until a megaflow returns without a SoftFlow action, or a loop detector triggers.

5.3 Actions

In addition to the SoftFlow datapath prototype, we developed several SoftFlow actions in an effort to evaluate both the efficacy of our design as well as to understand its implications to a service developer. As shortly discussed in the next section, these services range from a trivial packet counter, to an AES payload transcoder, connection tracker, and load balancer. There are several unique peculiarities to SoftFlow action development:

- SoftFlow instances are fine-grained and specific to a particular action configuration. A SoftFlow load-balancer, for example, can assume an incoming packet is TCP, is heading to a particular Virtual IP address, and runs over an HTTP port, because the OpenFlow flow table will guarantee it receives such packets. As a result, a service can be broken into smaller, more manageable, modules with simpler internals.
- SoftFlow services are based on the Open vSwitch code, and inherit the internal packet representation used throughout Open vSwitch. This internal representation contains pointers to the L2, L3, and L4 header offsets, saving SoftFlow actions the expense of re-parsing each packet.
- SoftFlow provides packets to service instances in batches. This allows implementations to utilize *prefetching* as per the DPDK guidelines: first prefetching the necessary internal state for a batch of packets into CPU cache, and only after that processing packets using that internal state. Such "staged" operation slightly complicates service implementations but is effective in reducing latency due to CPU cache misses.

6 Hardware Classification Offload

NIC vendors have a long history of introducing hardware acceleration techniques, with mixed success. TCP checksum and segmentation offload have proven useful, but a wide range of other NIC functions have been commercial failures. We contend this is due to an attempt to offload too much functionality. For instance, SR-IOV bypasses the virtual switch within hypervisors. While such an approach results in good performance, hardware is inflexible, has limited functionality, and has long update cycles that prevent it from adapting to new use cases. In this section, we discuss how, working cooperatively with the hardware, OVS can leverage classification offloads, and SoftFlow can take advantage of these offloads despite having *general* middlebox actions. This approach allows SoftFlow to benefit from the performance of hardware offload while maintaining the *generality* and *flexibility* of software forwarding.

Current commercially available high-end NICs already have a TCAM on-board, but its functionality is limited to QoS and Receive Side Scaling (RSS). For these use cases, the TCAM classification directs packets to different priority queues based on incoming L2–L4 headers [10]. Next-generation NICs including Intel's Boulder Rapids and Broadcom's BCM57300 NetXtreme C-Series [20] expand on this capability capability by modeling the TCAM as a generic OpenFlow-like switch.

While providing a programmable OpenFlow switch on a NIC partially alleviates flexibility concerns, a complete offload of switch functionality to the NIC presents challenges:

- **TCAM capacity.** TCAMs on NICs are quite limited in size and support only a fixed set of protocol header fields. This implies the maximum size of an offloaded OpenFlow table may be too small for practical classification offloading. Current NIC TCAMs have 16k entries or less, depending on the number of fields matched on.
- **Stateless operations only.** Executing OpenFlow actions on the NIC is enough for stateless L2–L4 operations, but stateful L4–L7 cannot be offloaded.

In SoftFlow, instead of offloading *all* classification to the NIC, we instead use the TCAM as a hardware flow cache *assisting* the software flow cache. This allows SoftFlow to offload the most active megaflows and leave the rest to software making TCAM limitations less of a concern. Similarly, since actions are performed by the CPU, limits on the actions available on the NIC are of no consequence.

To accelerate classification, we assign each megaflow a unique ID before pushing it into the NIC. Furthermore, we configure the NIC actions associated with each megaflow to write this unique ID into the metadata of each matching packet. If the NIC finds a match, software classification can be skipped, allowing SoftFlow to proceed to executing OpenFlow actions and SoftFlow services. Similarly, to work around matching limitations, SoftFlow can offload matches over the limited headers supported by the NIC, leaving the rest for software. We do not discuss this here further because it is explored deeply in SAX-PAC [14] and we expect future L2–L4 headers to be relatively well supported through more programmable packet parsing capabilities in future NICs.

While in general guaranteeing consistency of TCAM table updates is a potentially complicated topic [38], the priority-less eventually consistent design of the megaflow cache simplifies the problem. On reception of a packet with a TCAM hint from the NIC, we simply verify that the indicated megaflow still exists and the packet does, indeed, match it. Thus if the megaflow table changes in a way inconsistent with the NIC's TCAM, affected packets will simply fall back to software classification until the NIC is updated. Similarly, if a megaflow matches on fields

unsupported by the hardware, the discrepancy is resolved in software for misclassified packets.

Finally, while the offloading remains completely transparent to the service implementations – after all it is completely the responsibility of the SoftFlow datapath – the converse is not true. Services that cannot be coalesced require re-classification after execution. In SoftFlow these secondary classifications are not done at ingress, and therefore must be done in software. That is, SoftFlow cannot offload all packet classification to the NIC

7 Evaluation

In this section, we evaluate the benefits of SoftFlow's run-to-completion architecture, flow caching, and stage coalescing for workloads requiring complex middlebox services.

7.1 Test Environment

Our testbed has two identical servers each running a pair of 10-core Intel Haswell 2.6GHz CPUs with hyper-threading disabled, 25MB of L3 cache and 128GB RAM. Each server has an Intel 10Gb NIC with two ports for a total forwarding capacity of 20Gbps. The ports of each server are patched directly into the other with no switches or routers mediating them. One server is configured as a packet generator running Pktgen-DPDK [26], the other runs our SoftFlow prototype. In all tests, the SoftFlow test server receives packets, forwards them through a pipeline of OpenFlow tables and SoftFlow services, and forwards them back out the port on which they were received.

Except where noted, all experiments are run with a single SoftFlow core (as would be typical on a hypervisor), with coalescing enabled (for the actions that can take advantage of it), the TCAM simulator disabled (as initially most deployments won't have TCAM accelerated NICs), and a 100% megaflow cache hit rate after a brief ramp up period.

Packet traces. We evaluate three traces, which the traffic generator replays repeatedly at line rate on both ports for a total throughput of 20Gbps. The first trace, T1, was collected from a software network virtualization gateway appliance running Open vSwitch that was deployed in a private, production, multi-tenant datacenter servicing approximately one thousand hypervisors. The trace is 44 seconds long, contains 4.6 million packets, with an average packet size of 937 bytes, and average transmission rate of 795Mbps. We replay the trace at line rate and, on each replay, we replace the L4 source and destination ports with a consistent hash of their previous values to defeat the exact match flow cache. The trace contains 11k IP addresses participating in 100k distinct TCP conversations and 75k distinct UDP conversations. The maximum conversation length in our trace is 99k packets, the mean is 42 packets, and the median is 10 packets.

To further stress the prototype, we stripped the payload of T1 to simulate traffic with maximum packet size of 64 bytes. In our tests, the resulting packet trace is called "T2". In addition, to simulate ideal conditions, we generated a synthetic trace called "T3" consisting of a few long-lived, high volume connections– specifically, 32 transport connections, each in turn sending a burst of 256 packets of 64 bytes each.

7.2 Pipelines

We implemented four prototype SoftFlow pipelines to emulate use cases typical of network virtualization systems.

Pipeline A. Our first pipeline mimics a network virtualization gateway as described in [16]. It consists of four stages starting with 500 OpenFlow rules that match randomly generated Ethernet destination addresses. From there, it continues to an L3 table whose rules match 500 randomly generated IP prefixes. (Random flow tables like these are a worst case for Open vSwitch because they result in the most specific flow masks used in the flow cache [25].) Next is a SoftFlow firewall consisting of 500 ACLs and a SoftFlow `conntrack` action, as described in §4. The firewall is quite sophisticated as it maintains per-connection state, tracks TCP sequence numbers, and supports multiple protocols beyond TCP. The ACLs are a randomly chosen subset of a cloud provider's production firewall rule set. The firewall is followed by an L2 table, after which packets egress.

Pipeline B. The second pipeline demonstrates SoftFlow performance for a complex service chain of actions. The first stages are identical to Pipeline A: an L2/L3 lookup and stateful firewall. Following the firewall, we execute a content transcoder which encrypts the transport payload of each packet using AES-128 in CBC mode (similar to IPsec [13]). Like the firewall and load balancer, the content transcoder is implemented as a SoftFlow action. Finally, a stateless OpenFlow NAT moves the IP destination of the packets into the 10.0.0.0/8 prefix.

Pipeline C. The third pipeline demonstrates the additional cost of making forwarding decisions based on internal state. It is identical to Pipeline A except an implementation of the load balancer described in §4 is inserted just after the firewall. As discussed earlier, the load balancer defeats coalescing and therefore requires a megaflow lookup after each invocation. (In contrast, Pipelines A and B use coalescing.)

Pipeline D. Our final pipeline, D, is a configurable number of "no-op" SoftFlow actions designed to isolate action invocation overhead. For brevity, we only present results for this pipeline in a couple of cases that are particularly interesting.

7.3 Measurements

Run-to-Completion. A common approach to virtualizing middleboxes, both in practice and in the literature [7, 19, 24] is to allocate a dedicated virtual machine to each middlebox. This approach provides strong isolation between middleboxes, and a simple migration path from legacy implementations, but as we show in Figure 5, it comes at significant performance cost.

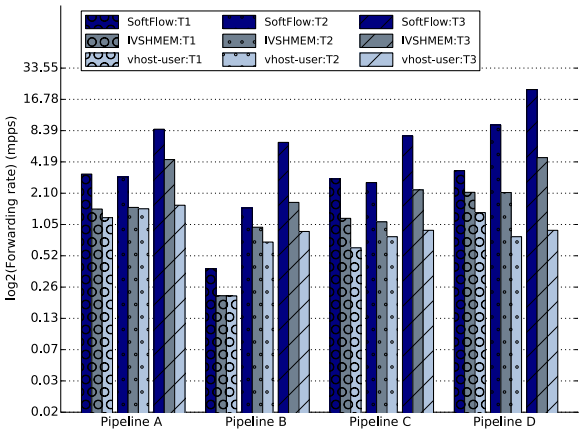


Figure 5: Comparison of forwarding rates (on a log scale) between SoftFlow and KVM virtual machines running one of two virtual NIC implementations: IVSHMEM and vhost-user.

We evaluate each of our test pipelines against equivalent virtual machine implementations in which each middlebox (*i.e.,* a SoftFlow action and its associated OpenFlow tables) is allocated its own dedicated SoftFlow VM: one for Pipeline A's firewall, two for Pipeline B (firewall and AES), two for Pipeline C (firewall and load balancer), and one for Pipeline D's `noop` action. The stateless L2 and L3 lookups not associated with a middlebox are implemented in the hypervisor vswitch (running the same SoftFlow implementation but without any SoftFlow actions). The hypervisor vswitch and the virtual machines are each allocated a dedicated CPU core for forwarding so, in total, Pipeline A runs 2 cores, Pipeline B runs 3, and Pipeline C runs 3.

We compare the VM implementation to a pure SoftFlow stack configured using run-to-completion forwarding. For each pipeline, the SoftFlow switch is allocated the same number of cores as the equivalent VM test so that performance comparisons are fair.

Conventional wisdom suggests that forwarding performance is determined by the virtual NIC. Therefore, we compare SoftFlow against two competitive DPDK virtual NIC drivers, IVSHMEM [11] and vhost-user [36]:

- *IVSHMEM*: a zero-copy implementation not dissimilar from the approach taken by NetVM [7]. This implementation relies on a shared memory region between hypervisor and guest over which pointers to packets are transferred. Note that the putative advantage of this approach is speed *at the cost of isolation* as virtual machines have read and write access to all packets on the hypervisor.
- *vhost-user*: the virtual NIC implementation officially recommended by Open vSwitch was developed in response to the limitations of IVSHMEM. The approach is similar to that taken by ClickOS [19] – packets are copied into and out of a shared memory region in the virtual machine address space. The overhead of each packet IO is the same as IVSHMEM except for an additional packet copy. In return for this copy, vhost-user achieves true isolation making it a suitable choice for untrusted virtual machines.

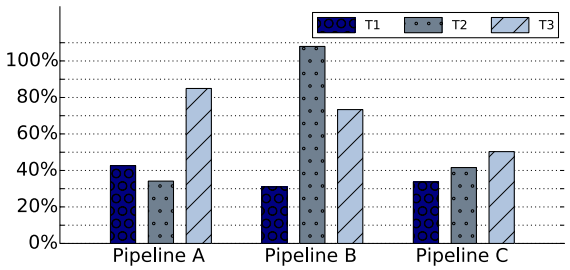


Figure 6: Percent improvement in single core forwarding throughput with stage coalescing enabled.

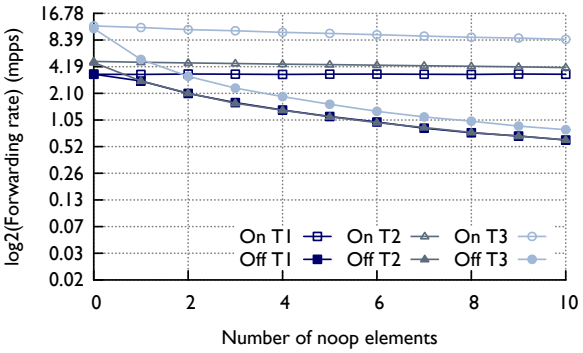


Figure 7: Single core forwarding rates (mpps) as a function of the number of `noop` service elements in Pipeline D. For each trace we test with coalescing enabled and disabled.

These two approaches represent a baseline similar to VM based NFV proposals like NetVM and ClickOS with their highly efficient virtual NIC implementations. IVSHMEM and vhost-user, are analogous to these approaches, but using OVS, thus allowing our results to isolate the cost of VM traversal versus the SoftFlow run-to-completion forwarding model.

As shown in Figure 5, SoftFlow consistently outperforms virtual machines by *2x* or more. Also, interestingly, the difference between IVSHMEM and vhost-user is less than we expected in many cases. For these cases, we believe the cost of VM traversal, (CPU cache misses, packet re-parsing, additional classification) outweights packet copies.

Stage coalescing. Figure 6 shows the percent improvement in forwarding rate caused by enabling classification coalescing for each pipeline and traffic pattern. While all cases benefit somewhat, the degree of the benefit is highly dependent on the pipeline and traffic pattern. Pipeline B, for instance, does AES encryption on the payload which mitigates the benefits of coalescing for T1's large packets, while for the small packets of T2 and T3, encryption is less dominant allowing coalescing to help more. Also, Pipeline C implements the `balancer` action which never coalesces packets, limiting the overall benefit. We also note that we

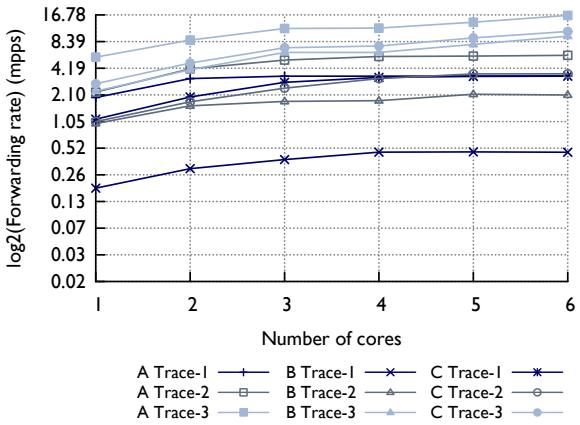


Figure 8: Performance as number of CPUs increases.

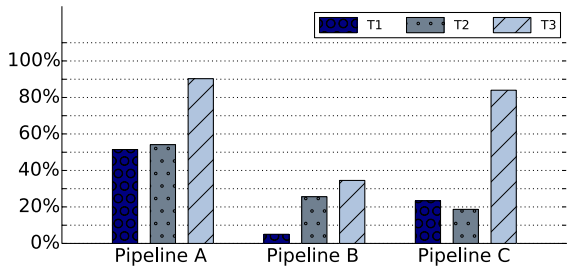


Figure 9: Percent improvement in single core forwarding throughput with TCAM offload and full stage coalescing enabled (see Figure 6).

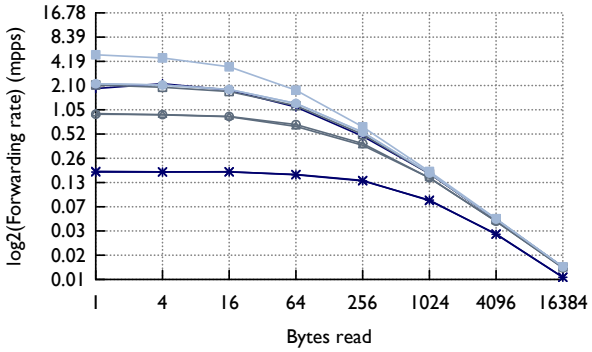


Figure 10: Single core forwarding rates when a service reads increasingly many random bytes of a 1GB block. See Figure 8 for the legend.

measured performance of coalescing as decreasing fractions of traffic were allowed to benefit, we've omitted the graph for brevity, but report that the benefit scales smoothly.

We additionally measure the effect of stage coalescing on Pipeline D. Figure 7 shows the packets per second forwarded with coalescing both enabled and disabled as the number of `noop` actions in the pipeline increases. Note that even for this trivial pipeline, traversing the cache hierarchy has real cost that can be avoided with stage coalescing.

Hardware offload. We do not yet have an engineering sample of a NIC with a programmable TCAM, so instead we emulated 1k TCAM entries in our existing testbed. Megaflow cache sizes witnessed with the above tests were rationale for this TCAM size: for instance, Pipelines A, B, and C resulted in 258, 21,335 and 256 active megaflows in coalesced tests with Trace 2. Tests in hypervisor virtual switch environments similarly suggest that a few hundred megaflows is sufficient for high hit rates [25]. We preprocessed the traces, encoding a rule ID for each connection in the IP ID field. Then we forwarded the packets through the prototype, and checked the field to choose a matching megaflow, thus allowing us to skip the megaflow classification. Figure 9 shows the percent improvement this feature provides. Note that the test was performed with stage coalescing enabled, so this benefit is on top of what coalescing can provide. The benefit depends highly on the pipeline and traffic pattern, ranging from 5% for Pipeline B (AES on large packets dominates), to 90% for the synthetic traces. We also measured fractional TCAM offload rates, and found the benefit to scale smoothly as offload rates approach 100%. We omit the graph for brevity.

Multi-core parallelism. The Open vSwitch datapath is designed to scale easily as NICs load-balance traffic across CPU cores, sharing little state. For this reason, and our focus on hypervisors where low CPU utilization is paramount, the majority of this evaluation has focused on single-core performance. However, in Figure 8, we did measure the affect of additional CPUs on performance. Both Pipeline A T1 and Pipeline C T1 quickly hit the limits of our 20Gbps test bed. The rest of the lines scale well, though sometimes hitting points of diminishing returns due to inefficiencies within the SoftFlow actions, or Open vSwitch itself.

Action Complexity. Packet processing on modern general-purpose processors is highly sensitive to memory access patterns. In an attempt to quantify the performance of complex SoftFlow actions which require traversing large complex data structures, we injected additional, synthetic memory load to the pipelines by introducing a service element that does nothing but make a series of random prefetched memory accesses. In Figure 10 we see how, regardless of successful flow caching, the throughput quickly drops as more random memory locations are accessed per packet.

Cache miss rate. Maintaining high flow cache hit rates is also critical for optimal throughput. While effective caching is the task of Open vSwitch flow cache architecture, we tested the overall impact of lowering cache hit rates for SoftFlow pipelines with L4–L7 services by synthetically introducing flow cache misses. Figure 11 validates the criticality of maintaining high cache hit rates for overall forwarding throughput.

CPU. Finally, we analyzed how the breakdown of CPU usage evolves if classification coalescing and TCAM

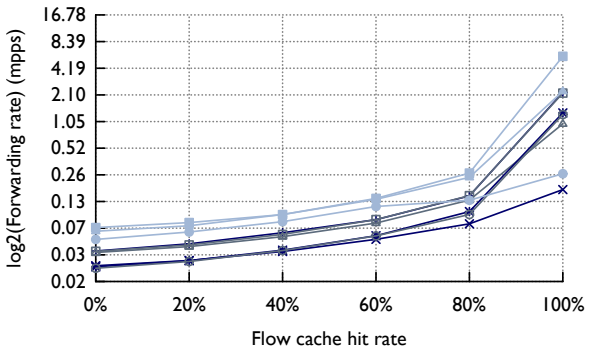


Figure 11: Single core forwarding rates as flow cache hit rate improves. At 0% the slow path handles all packets. See Figure 8 for the legend.

offloading are enabled with the Pipeline A and T2. In Figure 12, we see how both stage coalescing and hardware offloading reduce the portion of CPU used for megaflow lookup (the "Megaflow" column in the table), thus leaving more CPU resources for service execution. This naturally implies that the share of packet parsing and I/O grows.

8 Related Work

We now consider the related work in addition to the high-level approaches discussed in §2. First, we note that the interest in software forwarding was revived by the low-level hardware and software I/O optimizations that were identified to be necessary to substantially improve the ability of x86 to forward small packets at high rates [3, 8, 9, 27]. SoftFlow builds on the results of these efforts.

NFV and the requirements of datacenter workloads calling for high throughput networking capacity (both in packet rates and volume) have resulted in many privately and publicly documented optimizations in the virtual NIC I/O interface hypervisors provide for the VMs [7, 19, 28, 29]. Contrary to some of these efforts being used to integrate services as VMs, SoftFlow builds on tighter integration of packet processing across different service elements. As a result, while an unfair and incomplete comparison, SoftFlow is often able to reach line rate of 10Gbps even with one or two CPU cores, whereas VM based approaches use all the cores a server has to offer to accomplish the same with multi-stage pipelines [7, 19].

Perhaps the most closely related work is, therefore, the recent proposals arguing for the benefits of *vertical* integration of protocol stacks in userspace, to both improve performance [18] and accelerate innovation [6]. In SoftFlow, we take the integration a step further and demonstrate that also *horizontal* integration across all packet processing elements (L2–L7) can improve the overall performance.

An important alternative to the SoftFlow forwarding model was first proposed by Click [15], and adapted by ClickOS [19] for NFV workloads. This approach models switch processing as a graph of composable packet

	Megaflow	Services	Parsing	I/O
Base	57%	8%	6%	3%
Coalesce	44%	15%	5%	11%
TCAM	18%	28%	9%	20%

Figure 12: Approximate CPU utilization of various components for Pipeline A T2 with coalescing off, coalescing on, and coalescing combined with TCAM offload. Functions which could not be accounted to a particular column (*e.g.,* `memcmp()`) or which took less than 2% of CPU are omitted from the table.

processing elements. SoftFlow actions are analogous to Click elements, yet they operate in a very different context. SoftFlow only uses these actions for stateful processing, falling back to OpenFlow style match-action tables for everything else. This allows SoftFlow to take advantage of global flow caching, and classification coalescing, both of which would be impossible in the Click forwarding model. For packet classification heavy workloads, like network virtualization, this makes SoftFlow an ideal fit.

Open vSwitch has taken some steps to add common middlebox functionality, most notably with the addition of the `conntrack` action that allows packets to access a connection tracker within the datapath. Due to the lack of a framework like SoftFlow, this work requires significant manual development effort for each new action touching all parts of the OVS code base. Additionally, these efforts do not take advantage of SoftFlow classification coalescing and, thus, require unnecessary megaflow lookups.

There has been some work offloading the entire Open vSwitch datapath to an NPU accelerated NIC [21]. This design does not work in cooperation with middleboxes as SoftFlow does. On the other hand, the SoftFlow hardware offload design only offloads expensive packet classification, leaving complex action processing to the CPU.

While, there has been significant work building integrated middleboxes on commodity systems [1, 4, 24, 30], we are the first system to specifically target the vast deployment of OpenFlow based network virtualization platforms with an architecture that's practically deploy-able, and tightly integrated with stateless L2–L3 services.

9 Conclusion

In this paper we described the design of SoftFlow, an extension of Open vSwitch designed to bring tightly integrated middleboxes to network virtualization platforms. Contrary to traditional software datapath designs, SoftFlow integrates network services tightly together to facilitate pervasive flow caching, removal of redundant packet classifications through stage coalescing, and the use of hardware classification offloads. These advantages coupled with SoftFlow's run-to-completion forwarding model allow it to significantly outperform virtual machine based alternatives, while being a better fit for existing Open vSwitch deployments.

References

[1] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proc. of ANCS*, 2012.

[2] Daniele Di Proietto. `https://github.com/ddiproietto/ovs/tree/userconntrack_20150908`.

[3] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. of SOSP*, 2009.

[4] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward Software-Defined Middlebox Networking. In *Proc. of HotNets*, 2012.

[5] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proc. of SIGCOMM*, August 1999.

[6] M. Honda, F. Huici, C. Raiciu, J. Araújo, and L. Rizzo. Rekindling Network Protocol Innovation with User-level Stacks. *SIGCOMM CCR*, 44(2):52–58, 2014.

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

[8] Intel Data Direct I/O Technology, 2013. `http://www.intel.com/content/www/us/en/io/direct-data-i-o.html`.

[9] Intel DPDK: Data Plane Development Kit, 2013. `http://dpdk.org`.

[10] Intel Ethernet Flow Director. `http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-flow-director-video.html`, September 2015.

[11] IVSHMEM Libarary. `http://dpdk.org/doc/guides/prog_guide/ivshmem_lib.html`.

[12] Justin Pettit and Ben Pfaff and Chris Wright and Madhu Venugopal. OVN, Bringing Native Virtual Networking to OVS.

[13] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, IETF, December 2005.

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

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[16] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, 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*, 2014.

[17] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proc. of ASPLOS*, 2013.

[18] I. Marinos, R. N. Watson, and M. Handley. Network Stack Specialization for Performance. In *Proc. of HotNets*, 2013.

[19] 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*, 2014.

[20] `https://www.broadcom.com/press/release.php?id=s923886`.

[21] R. Neugebauer. Selective and Transparent Acceleration of OpenFlow Switches (Netronome Whitepaper), July 2014. `https://netronome.com/wp-content/uploads/2014/07/Netronome-Selective-and-Transparent-Acceleration-of-OpenFlow-Switches-Whitepaper_4-13.pdf`.

[22] Network Virtualization Platform. `http://www.nicira.com/en/network-virtualization-platform`.

[23] Open vSwitch – An Open Virtual Switch. `http://www.openvswitch.org`, September 2015.

[24] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proc. Symposium on Operating Systems Principles*, October 2015.

[25] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proc. of NSDI*, 2015.

[26] Pktgen Traffic Generator Using DPDK. `https://github.com/Pktgen/Pktgen-DPDK/`, September 2015.

[27] L. Rizzo. Netmap: a Novel Framework for Fast Packet I/O. In *Proc. of USENIX ATC*, June 2012.

[28] L. Rizzo and G. Lettieri. VALE, a Switched Ethernet for Virtual Machines. In *Proc. of CoNEXT*, 2012.

[29] L. Rizzo, G. Lettieri, and V. Maffione. Speeding Up Packet I/O in Virtual Machines. In *Proc. of ANCS*, 2013.

[30] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. of NSDI*, 2012.

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

[32] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. of SIGCOMM*, 2012.

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

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

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

[36] Vhost Libarary. `http://dpdk.org/doc/guides/prog_guide/vhost_lib.html`.

[37] VMware NSX. `http://www.vmware.com/products/nsx`, September 2015.

[38] Z. Wang, H. Che, M. Kumar, and S. K. Das. CoPTUA: Consistent Policy Table Update Algorithm for TCAM Without Locking. *IEEE Trans. Comput.*, 53(12), December 2004.