# VFP: A Virtual Switch Platform for Host SDN in the Public Cloud

Daniel Firestone, *Microsoft*

## Abstract

Many modern scalable cloud networking architectures rely on host networking for implementing VM network policy - e.g. tunneling for virtual networks, NAT for load balancing, stateful ACLs, QoS, and more. We present the Virtual Filtering Platform (VFP) - a programmable virtual switch that powers Microsoft Azure, a large public cloud, and provides this policy. We define several major goals for a programmable virtual switch based on our operational experiences, including support for multiple independent network controllers, policy based on connections rather than only on packets, efficient caching and classification algorithms for performance, and efficient offload of flow policy to programmable NICs, and demonstrate how VFP achieves these goals. VFP has been deployed on >1M hosts running IaaS and PaaS workloads for over 4 years. We present the design of VFP and its API, its flow language and compiler used for flow processing, performance results, and experiences deploying and using VFP in Azure over several years.

## 1. Introduction

The rise of public cloud workloads, such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform [13-15], has created a new scale of datacenter computing, with vendors regularly reporting server counts in the millions. These vendors not only have to provide scale and the high density/performance of Virtual Machines (VMs) to customers, but must provide rich network semantics, such as private virtual networks with customer supplied address spaces, scalable L4 load balancers, security groups and ACLs, virtual routing tables, bandwidth metering, QoS, and more.

This policy is sufficiently complex that it often cannot economically be implemented at scale in traditional core routers and hardware. Instead a common approach has been to implement this policy in software on the VM hosts, in the virtual switch (vswitch) connecting VMs to the network, which scales well with the number of servers, and allows the physical network to be simple, scalable and very fast. As this model separates a centralized control plane from a data plane on the host, it is widely considered an example of Software Defined Networking (SDN) – in particular, host-based SDN.

As a large public cloud provider, Azure has built its cloud network on host based SDN technologies, using them to implement almost all virtual networking features we offer. Much of the focus around SDN in recent years has been on building scalable and flexible network controllers and services, which is critical. However, the design of the programmable vswitch is equally important. It has the dual and often conflicting requirements of a highly programmable dataplane, with high performance and low overhead, as cloud workloads are cost and performance sensitive.

In this paper, we present the Virtual Filtering Platform, or VFP – our cloud scale virtual switch that runs on all of our hosts. VFP is so named because it acts as a filtering engine for each virtual NIC of a VM, allowing controllers to program their SDN policy. Our goal is to present both our design and our experiences running VFP in production at scale, and lessons we learned.

### 1.1 Related Work

Throughout this paper, we use two motivating examples from the literature and demonstrate how VFP supports their policies and actions. The first is VL2 [2], which can be used to create virtual networks (VNETs) on shared hardware using stateless tunneling between hosts. The second is Ananta [4], a scalable Layer-4 load balancer, which scales by running the load balancing NAT in the vswitch on end hosts, leaving the in-network load balancers stateless and scalable.

In addition, we make references and comparisons to OpenFlow [5], a programmable forwarding plane protocol, and OpenVswitch [1] (OVS), a popular open source vswitch implementing OpenFlow. These are two seminal projects in the SDN space. We point out core design differences from the perspective of a public cloud on how our constraints can differ from those of open source projects. It is our goal to share these learnings with the broader community.

## 2. Design Goals and Rationale

VFP's design has evolved over time based on our experiences running a large public cloud platform. VFP was not our original vswitch, nor were its original functions novel ideas in host networking – VL2 and Ananta already pioneered such use of vswitches.

Originally, we built networking filter drivers on top of Windows's Hyper-V hypervisor for each host function, which we chained together in a vswitch – a stateful firewall driver for ACLs, a tunneling driver for VL2 VNETs, a NAT driver for Ananta load balancing, a QoS driver, etc. As host networking became our main tool for virtualization policy, we decided to create VFP in 2011 after concluding that building new fixed filter drivers for host networking functions was not scalable

or desirable. Instead, we created a single platform based on the Match-Action Table (MAT) model popularized by projects such as OpenFlow [5]. This was the origin of our VFPAPI programming model for VFP clients.

VFP's core design goals were taken from lessons learned in building and running both these filters, and the network controllers and agents on top of them.

### 2.1 Original Goals

The following were founding goals of the VFP project:

1. *Provide a programming model allowing for multiple simultaneous, independent network controllers to program network applications, minimizing cross-controller dependencies.*

Implementations of OpenFlow and similar MAT models often assume a single distributed network controller that owns programming the switch (possibly taking input from other controllers). Our experience is that this model doesn't fit cloud development of SDN – instead, independent teams often build new network controllers and agents for those applications. This model reduces complex dependencies, scales better and is more serviceable than adding logic to existing controllers. We needed a design that not only allows controllers to independently create and program flow tables, but would enforce good layering and boundaries between them (e.g. disallow rules to have arbitrary GOTOs to other tables, as in OpenFlow) so that new controllers could be developed to add functionality without old controllers needing to take their behavior into account.

2. *Provide a MAT programming model capable of using connections as a base primitive, rather than just packets – stateful rules as first class objects.*

OpenFlow's original MAT model derives historically from programming switching or routing ASICs, and so assumes that packet classification must be stateless due to hardware resources available. However, we found our controllers required policies for connections, not just packets – for example end users often found it more useful to secure their VMs using stateful Access Control Lists (ACLs) (e.g. allow outbound connections, but not inbound) rather than stateless ACLs used in commercial switches. Controllers also needed NAT (e.g. Ananta) and other stateful policies. Stateful policy is more tractable in soft switches than in ASIC ones, and we believe our MAT model should take advantage of that.

3. *Provide a programming model that allows controllers to define their own policy and actions, rather than implementing fixed sets of network policies for predefined scenarios.*

Due to limitations of the MAT model provided by OpenFlow (historically, a limited set of actions, limited rule scalability and no table typing), OpenFlow switches such as OVS have added virtualization functionality outside of the MAT model. For example, constructing virtual networks is accomplished via a virtual tunnel endpoint (VTEP) schema [29] in OVSDB [8], rather than rules specifying which packets to encapsulate (encap) and decapsulate (decap) and how to do so.

We prefer instead to base all functionality on the MAT model, trying to push as much logic as possible into the controllers while leaving the core dataplane in the vswitch. For instance, rather than a schema that defines what a VNET is, a VNET can be implemented using programmable encap and decap rules matching appropriate conditions, leaving the definition of a VNET in the controller. We've found this greatly reduces the need to continuously extend the dataplane every time the definition of a VNET changes.

The P4 language [9] attempts a similar goal for switches or vswitches [38], but is very generic, e.g. allowing new headers to be defined on the fly. Since we update our vswitch much more often than we add new packet headers to our network, we prefer the speed of a library of precompiled fast header parsers, and a language structured for stateful connection-oriented processing.

### 2.2 Goals Based on Production Learnings

Based on lessons from initial deployments of VFP, we added the following goals for VFPv2, a major update in 2013-14, mostly around serviceability and performance:

4. *Provide a serviceability model allowing for frequent deployments and updates without requiring reboots or interrupting VM connectivity for stateful flows, and strong service monitoring.*

As our scale grew dramatically (from O(10K) to O(1M) hosts), more controllers built on top of VFP, and more engineers joined us, we found more demand than ever for frequent updates, both features and bug fixes. In Infrastructure as a Service (IaaS) models, we also found customers were not tolerant of taking downtime for individual VMs for updates. This goal was more challenging to achieve with our complex stateful flow model, which is nontrivial to maintain across updates.

5. *Provide very high packet rates, even with a large number of tables and rules, via extensive caching.*

Over time we found more and more network controllers being built as the host SDN model became more popular, and soon we had deployments with large numbers of flow tables (10+), each with many rules, reducing performance as packets had to traverse each table. At the same time, VM density on hosts was increasing, pushing us from 1G to 10G to 40G and even faster NICs. We needed to find a way to scale to more policy without impacting performance, and concluded we needed to perform compilation of flow actions

across tables, and use extensive flow caching, such that packets on existing flows would match precompiled actions without having to traverse tables. *Provide a fast packet classification algorithm for cases with large numbers of rules and tables.*

While solving Goal #5 dramatically improved performance for existing flows (e.g. all TCP packets following a SYN), we found a few applications pushing many thousands of rules into their flow tables (for example, a distributed router BGP peering with customers, using VFP as its FIB), which slowed down our flow compiler. We needed to design an efficient packet classifier to handle performance for these cases.

6. *Implement an efficient mechanism to offload flow policy to programmable NICs, without assuming complex rule processing.*

As we scaled to 40G+ NICs, we wanted to offload policy to NICs themselves to support SR-IOV [22, 23] and let NICs indicate packets directly to VMs while applying relevant VFP policy. However, as controllers created more flow tables with more rules, we concluded that directly offloading those tables would require prohibitively expensive hardware resources (e.g. large TCAMs, matching in series) for server-class NICs. So instead of trying to offload classification operations, we wanted an offload model that would work well with our precompiled exact-match flows, requiring hardware to only support accessing a large table of cached flows in DRAM, and support for our associated action language.

### 2.3 Non-Goals

The following are goals we've seen in other projects, which based on our experiences we chose not to pursue:

1. *Providing cross-platform portability.*

Portability is difficult to achieve with a high performance datapath in the kernel. Projects such as OVS have done this by splitting into a kernel fastpath and a portable userspace slowpath with policy – but this comes at a cost of over an order of magnitude slowdown when packets take the slowpath [16]. We run entirely on one host OS, so this wasn't a goal for us.

2. *Supporting a network / remote configuration protocol bundled with VFP itself.*

OpenFlow contains both a network programming model, as well as a wire protocol for configuring rules over the network. The same is true of OVS and the OVSDB protocol. In order to enable different controller models of managing policy (e.g. a rule push model, or the VL2 Directory System pull model), we instead decoupled VFP as a vswitch from the agents that implement network configuration protocols, and focused on providing a high performance host API.

3. *Providing a mechanism to detect or prevent controllers from programming conflicting policy*

Much literature [17-21] describes attempts to detect or prevent conflicts or incorrect policy in flow table or rule matching systems. Despite our first goal of supporting multiple controllers programming VFP in parallel without interfering with each other, we concluded early on that explicit conflict management was neither a feasible nor necessary goal, for several reasons. Programming VFP on behalf of a VM is a protected operation that only our controllers can perform, so we are not worried about malicious controllers. In addition, we concluded it was impossible to tell the difference between a misprogrammed flow table overwriting another flow table's actions by accident, and a flow table designed to filter the output of another table. Instead we focused on tooling to help developers validate their policy and interactions with other policies.

## 3. Overview and Comparison

As a motivating example throughout the paper, we consider a simple scenario requiring 4 host policies used for O(1M) VMs in a cloud. Each policy is programmed by its own SDN controller and requires both high performance and SR-IOV offload support: A VL2-style VNET, an Ananta-style load balancer, a stateful firewall, and per-destination traffic metering for billing purposes. We begin by evaluating this against existing solutions to demonstrate the need for a different approach, which we describe. Sections 4-7 then detail VFP's core design.

### 3.1 Existing solutions: Open vSwitch

While Linux and Windows support bridging [26-28] between multiple interfaces, which can be used as a vswitch, these bridges don't apply SDN policy. Other public clouds such as Google have described [25] using host SDN policy, but details are not public. OVS is the primary solution today to provide vswitch-based SDN, and so (as of version 2.5) is our main comparison point.

We believe OVS has had a great positive impact in making programmable host networking widely available. Many OVS design choices were driven by OVS-specific goals such as cross-platform support and the requirements of shipping in the Linux kernel[1] [1]. Combined with OVS's use of OpenFlow, these designs enable deployments with controllers managing virtual switches and physical switches via the same protocols, which was a non-goal for our host-based networking model. OVS also supports many protocols useful for physical switches such as STP, SPBM, BFD, and

[1] Windows is always backwards compatible with drivers, so we can ship a single driver compatible with all recent Windows versions without needing kernel integration.

IGMP Snooping [3], that we don't use.

Partially as a result of OpenFlow in particular, however, aspects of OVS make it unsuitable for our workload:

- OVS doesn't natively support true independent multi-controller models, as is required when our VL2 and Ananta applications are controlled separately. The underlying OpenFlow table model is unsuitable for multi-controller use cases – table rules specify explicit GOTOs to next tables, causing controllers to tie their policy together. Also, tables can only be traversed in the forward direction, whereas multi-controller scenarios require packets to traverse tables in the reverse direction for outbound packets as for inbound, so that packets will be in a consistent state when matching that controller's policy in either direction. VFP solves this with explicit table layering (§5.2).
- OVS doesn't natively support stateful actions like NAT in its MAT model, required by our Ananta example (our firewall is stateful too) – in both cases controllers need to operate on connections as a base primitive rather than packets. OpenFlow provides only for a packet model, however. OVS recently added support for sending packets to the Linux connection tracker to enable stateful firewall, but it's not exposed as a MAT and doesn't easily support a NAT, which requires explicit bidirectional stateful tables so that the NAT is reversed on a flow's return path. VFP solves this with stateful layers (§5.2).
- OVS's VTEP Schema requires explicit tunnel interfaces to implement VL2-style VNETs rather than allowing the controller to specify its own encap / decap actions, which aren't natively supported in OpenFlow[2]. This hardcodes a model of a VNET in the dataplane rather than allowing the controller to define how the VNET works (Goal 3). Adding complex VNET logic like ECMP routing can be difficult in this schema and requires vswitch changes, rather than policy changes. VFP supports all of these directly in its MAT (§5.3) by modeling encap/decap as actions.
- OVS doesn't support a VL2-style Directory System, required to dynamically look up Customer Address to Physical Address mappings. OpenFlow's design lacks the scalability to support large VNETs this way – OpenFlow exception packets must all go back to the central controller, and in OVS, VTEPs on all hosts are expected to be updated any time a mapping changes. This is OK for NSX/vSphere, which support up to 1000 hosts [30], but we found this unusable at our scale. VFP solves this by combining the schema-free MAT model with efficient asynchronous I/O exception requests (§5.5.1) that an agent can redirect to services separate from the controller.
- OVS doesn't have a generic offload action language or API that can support combinations of policy such as an Ananta NAT plus a VL2 encap. While SR-IOV offloads have been implemented on top of OVS builds by NIC vendors for specific workloads (such as VTEP schema) [31], doing general purpose offloads requires hardware to support the complex multi-table lookups of the original policy (e.g. [32]) that we've found quite costly in practice. VFP's Header Transposition language (§6.1.2, 9.3) enables SR-IOV support for all policy with only a single table lookup in hardware.

Thus we need a different design for our policy.

[2] While OpenFlow can support header pushes like MPLS tags as an action, it doesn't work for VNETs, e.g. VXLAN.

### 3.2 VFP Design

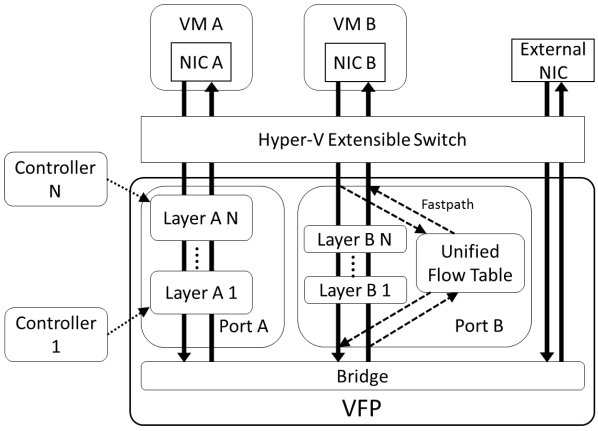


**Figure 1. Overview of VFP Design**

Figure 1 shows a model of the VFP design, which is described in subsequent sections. VFP operates on top of Hyper-V's extensible switch, as described in Section 4's filtering model. VFP implements MATs as layers that support a multi-controller model, with a programming model presented in Section 5. Section 6 describes VFP's packet processor, including a fastpath through unified flow tables and a classifier used to match rules in the MAT layers. Section 7 presents the switching model of VFP's bridge.

## 4. Filtering Model

VFP filters packets in the OS through MAT flow table policy. The filtering model is described below.

### 4.1 Ports and NICs

The core VFP model assumes a switch with multiple ports which are connected to virtual NICs (VNICs). VFP filters traffic from a VNIC to the switch, and from the switch to a VNIC. All VFP policy is attached to a specific port. From the perspective of a VM with a VNIC attached to a port, ingress traffic to the switch is

considered to be "outbound" traffic from the VM, and egress traffic from the switch is considered to be "inbound" traffic to the VM. VFPAPI and its policies are based on the inbound/outbound model.

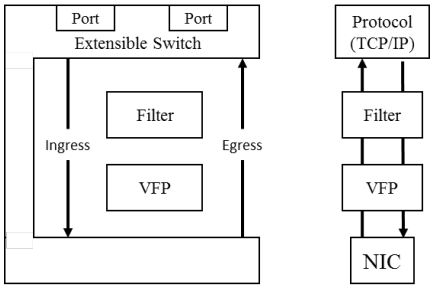


**Figure 2. Hyper-V Switch Extensibility vs NIC Filtering**

VFP implements a switch abstraction interface to abstract out different environments, instantiations of which provide logic for management (e.g. create / delete / connect / disconnect) of ports, VNICs, and associated objects. This interface supports both a Hyper-V switch and a filter for native hosts, shown in Figure 2.

### 4.2 Hyper-V Switch Extensibility

Hyper-V includes a basic vswitch [28] to bridge VNICs to a physical NIC. The switch is extensible, allowing filters to plug in and filter traffic to and from VNICs.

VFP acts as a Forwarding Extension to Hyper-V's vswitch – it simply replaces the entire switch logic with itself. Using this model allows us to keep our policy module and virtual switching logic (VFP) separate from the Hyper-V infrastructure to deliver packets to and from VMs, improving modularity and serviceability.

VFP in this mode supports PacketDirect [11], which allows a client to poll a NIC with very low overhead.

## 5. Programming Model

VFP's core programming model is based on a hierarchy of VFP objects that controllers can create and program to specify their SDN policy. The objects are:

- *Ports*, the basic unit that VFP policy filters on.
- *Layers*, the stateful flow tables that hold MAT policy.
- *Groups*, entities to manage and control related groups of rules within a layer.
- *Rules*, the match action table entries themselves.

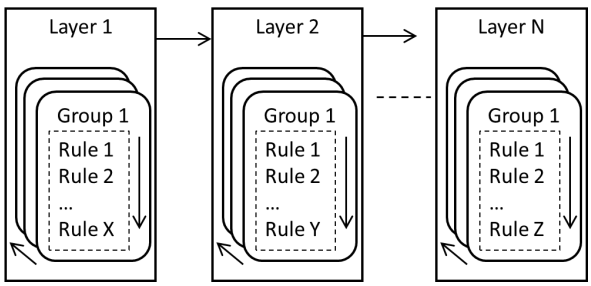


**Figure 3. VFP Objects: Layers, Groups, and Rules**

### 5.1 Ports

VFP's policy is implemented on a per-port basis – each port has match action tables which can sit on the inbound or outbound path of the port, acting as filters. Since our controllers generally want to program policy on behalf of a VM or VNIC, this clean separation of ports allows controllers to independently manage policy on different VMs, and instantiate and manage flow tables only on ports where they are needed – for example a VM in a virtual network may have tables to encapsulate and decapsulate traffic into tunnels, which another VM not in a virtual network wouldn't need (the VNET controller may not even be aware of the other VM, which it doesn't need to manage).

Policy objects on VFP are arranged in fixed object hierarchies, used to specify which object a given API call is referencing, such as Layer/Group/Rule. All objects are be programmed with a priority value, in which order they will be processed by rule matching.

### 5.2 Layers

VFP divides a port's policy into layers. Layers are the basic Match Action Tables that controllers use to specify their policy. They can be created and managed separately by different controllers, or one controller can create several layers. Each layer contains inbound and outbound rules and policies that can filter and modify packets. Logically, packets go through each layer one by one, matching rules in each based on the state of the packet after the action performed in the previous layer. Controllers can specify the ordering of their layers in a port's pipeline with respect to other layers, and create and destroy layers dynamically during operation.

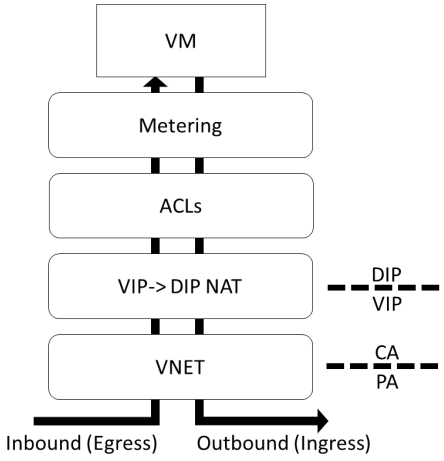


**Figure 4. Example VFP Layers with Boundaries**

Critically, packets traverse layers in the opposite order when inbound than when outbound. This gives them a "layering" effect when controllers implement opposite policy on either side of a layer. Take for example a load balancing layer implementing the Ananta NAT design.

On the inbound direction, the layer NATs connections destined to a Virtual IP (a VIP) to a Direct IP (DIP) behind the VIP – in this case the IP of the VM. On the outbound direction, it NATs packets back from DIP to VIP. The layer thus implements an address space boundary – all packets above it are in "DIP Space", and all packets below it are in "VIP Space". Other controllers can choose to create layers above or below this NAT layer, and can plumb rules to match VIPs or DIPs respectively – all without coordination with or involvement of the NAT controller.

Figure 4 shows layers for our SDN deployment example. VL2 is implemented by a VNET layer programmed by a virtual network controller, using tunneling for Customer Addresses (CAs) so that packets can traverse a physical network in Physical Address (PA) space recognized by physical switches in the path between VMs. This layer creates a CA / PA boundary by having encapsulation rules on the outbound path and decapsulation rules in the inbound path. In addition, an ACL layer for a stateful firewall sits above our Ananta NAT layer. The security controller, having placed it here with respect to those boundaries, knows that it can program policies matching DIPs of VMs, in CA space. Finally a metering layer used for billing sits at the top next to the VM, where it can meter traffic exactly as the customer in the VM sees it – all traffic that made it in and all traffic that was sent out from the VM.

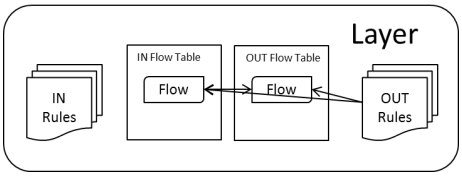


**Figure 5. A Layer with a stateful flow**

Layering also gives us a good model on which to implement stateful policy. Since packets on a given connection should be in the same IP/Port state on both the inbound and outbound path, we can keep flow state on a layer by assuming that a TCP or UDP 5-tuple (SrcIP, DstIP, IP Protocol, SrcPort, DstPort) will be the opposite on each side of the layer, and encoding that in a hash table of all connections in either direction. When a stateful rule is matched, it creates both an inbound and outbound flow in the layer flow tables, with the flow in the direction of the rule having the action of the rule, and the opposite direction taking the opposite action, to maintain layering. These inbound and outbound flows are considered *paired* – their actions simply change the packet to the state of the opposite flow in the pair rather than carrying their own action context.

When processing packets, VFP searches for a single rule in each layer to match by searching the groups of rules inside a layer for a matching rule. That rule's action is then performed on the packet – only one rule can match a given packet in a given layer (other matching rules of lower priority are ignored).

### 5.3 Rules

Rules are the entities that perform actions on matching packets in the MAT model. Per Goal #3, rules allow the controller to be as expressive as possible while minimizing fixed policy in the dataplane. Rules are made up of two parts: a condition list, specified via a list of conditions, and an action, both described below.

#### *5.3.1 Conditions*

When a VFPAPI client programs a rule, it provides a descriptor with a list of conditions. Conditions have a type (such as source IP address), and a list of matching values (each value may be a singleton, range, or prefix). For a condition to match a packet, any of the matching values can match (an OR clause). For a rule to match, all conditions in the rule much match (an AND clause).

#### *5.3.2 Actions*

A rule descriptor also has an action. The action contains a type and a data structure specific to that type with data needed to perform the rule (for example, an encapsulation rule takes as input data the source / destination IP addresses, source / destination MACs, encapsulation format and key to use in encapsulating the packet). The action interface is extensible - example conditions and actions are listed in Figure 6.

Rules are implemented via a simple callback interface (Initialize, Process Packet, Deinitialize) so as to make the base VFP platform easily extensible. If a rule type supports stateful instantiation, the process handler will create a pair of flows in the layer as well – flows are also typed and have a similar callback interface to rules. A stateful rule includes a flow time to live, which is the time that flows it creates will remain in the flow table after the last packet matches (unless expired explicitly by the TCP state machine described in §6.4.2).

| Conditions | Actions |
|---|---|
| Source/Dest MAC | Allow/Block (Stateful/Stateless) |
| Source/Dest IP | |
| Source/Dest TCP Port | NAT (L3/L4), (Stateful/Stateless) |
| Source/Dest UDP Port | |
| GRE Key | Encap/Decap |
| VXLAN VNI | QoS – Rate Limit, Mark DSCP, Meter |
| VLAN ID | Encrypt/Decrypt |
| Metadata From Previous Layer | Stateful Tunneling |
| | Routing (ECMP) |

**Figure 6. Example Conditions and Actions**

#### *5.3.3 User Defined Actions*

In addition to a large set of actions we'd created over

time, in VFPv2 we added user-defined actions to further Goal #3 – allowing the controllers to create their own rule types using a language for header field manipulations (Header Transpositions, see §6.1.2). This allows extending the base VFP action set without writing code to implement an action in the datapath.

### 5.4 Groups

Rules on a layer are organized into logical groups for management purposes. Groups are the atomic unit of policy in VFP – clients can transactionally update them. When classifying packets, VFP iterates through groups in a layer to find the highest priority rule in each group that matches the packet. By default, VFP will select the rule matched by the last group in the list. A rule can be marked "terminating," meaning that if it ever matches it will be applied immediately without traversing further groups. Groups can have conditions just like rules – if a group's condition doesn't match, VFP will skip it.

Below are two examples of how we've seen groups used for management of different policies in one layer:

- For VMs with Docker-style containers [35], each with its own IP, groups can be created and managed on a per-container basis by setting an IP condition on them.
- For our stateful firewall, infrastructure ACLs and customer ACLs can be expressed as two groups in a layer. Block rules would be marked terminating – if either group blocks it, a packet is dropped. Only if both sets of rules allowed a packet does it go through.

In addition to priority-based matching, individual groups can be Longest Prefix Matching on a condition type (for example, destination IP) to support routing scenarios. This is implemented as a compressed trie.

### 5.5 Resources

MATs are a good model for programming general network policy, but on their own aren't optimal for every scenario, especially ones with exception events. VNET requires a CA->PA lookup on outbound traffic (using a Directory System). Rules alone aren't optimal for such large mapping tables. So we support an extensible model of generic resources – in this case, a hash table of mappings. A resource is a port-wide structure that any rule on a port can reference. Another example is a range list, which can implement a dynamic source NAT rule of the form described in Ananta.

#### *5.5.1 Event Handling / Lookups*

Fast eventing APIs are required for many SDN applications where there is a lookup miss. We generally handle events in the context of resources – e.g. if an encap rule looks up a PA/CA mapping resource and misses, a VFPAPI client can register an efficient callback mechanism using async I/O and events. We use the same mechanism for Ananta NAT port exhaustion.

## 6. Packet Processor and Flow Compiler

As we scaled production deployments of VFP, and SDN became more widely used, it became necessary to write a new VFP datapath for improved performance and scalability across many rules and layers. Our work to improve performance, without losing the flexibility and programmability of VFPAPI, is described below.

### 6.1 Metadata Pipeline Model

VFP's original 2012 release, while performant under the workloads it was designed for, didn't scale well when the host SDN model took off even faster than we expected and many new layers were created by controllers. VFP rules and flows were implemented as callbacks which took a packet as input and modified its buffer - the next layer would have to reparse it. The original rule classification logic was linear match (with stateful flows accelerating this). At 10+ layers with thousands of rules, we needed something better.

A primary innovation in VFPv2 was the introduction of a central packet processor. We took inspiration from a common design in network ASIC pipelines e.g. [34] – parse the relevant metadata from the packet and act on the metadata rather than on the packet, only touching the packet at the end of the pipeline once all decisions have been made. We compile and store flows as we see packets. Our just-in-time flow compiler includes a parser, an action language, an engine for manipulating parsed metadata and actions, and a flow cache.

#### *6.1.1 Unified FlowIDs*

VFP's packet processor begins with parsing. The relevant fields to parse are all those which can be matched in conditions (from §5.3.1). One each of an L2/L3/L4 header (as defined in table 1) form a header group, and the relevant fields of a header group form a single FlowID. The tuple of all FlowIDs in a packet is a Unified FlowID (UFID) – the output of the parser.

#### *6.1.2 Header Transpositions*

Our action primitives, Header Transpositions (HTs), so called because they change or shift fields throughout a packet, are a list of paramaterizable header actions, one for each header. Actions (defined in table 2) are to *Push* a header (add it to the header stack), *Modify* a header (change fields within a given header), *Pop* a header (remove it from the header stack), or *Ignore* a header (pass over it). HTs are parameterized with all fields in a given header that can be matched (so as to create a complete language – any valid VFP flow can be transformed into any other valid VFP flow via exactly one HT). Actions in a HT are grouped into header groups. Table 3 shows examples of a NAT HT used by Ananta, and encap/decap HTs used by VL2.

As part of VFPv2, all rule processing handlers were

updated to take as input a FlowID and output a transposition. This has made it easy to extend VFP with new rules, since implementing a rule doesn't require touching packets – it's a pure metadata operation.

**Table 1. Valid Parameters for Each Header Type**

| Header | Parameters |
|---|---|
| Ethernet (L2) | Source MAC, Dest MAC |
| IP (L3) | Source IP, Dest IP, ToS (DSCP+ECN) |
| Encapsulation (L4) | Encapsulation Type, Tenant ID, Entropy (Optional) |
| TCP/UDP (L4) | Source Port, Dest Port, TCP Flags (note: does not support Push/Pop) |

**Table 2. Header Transposition Actions**

| Action | Notes |
|---|---|
| **Pop** | Remove this header. |
| **Push** | Push this header onto the packet. All header parameters for creating the new header are specified. |
| **Modify** | Modify this header. All header parameters needed are optional, but at least one is specified. |
| **Ignore** | Leave this header as is. |

**Table 3. Example Header Transpositions**

| Header | NAT | Encap | Decap | Encap+ NAT |
|---|---|---|---|---|
| Outer Ethernet | Ignore | Push (SMAC, DMAC) | Pop | Push (SMAC, DMAC) |
| Outer IP | Modify (SIP,DIP) | Push (SIP,DIP) | Pop | Push (SIP,DIP) |
| GRE | Not Present | Push (Key) | Pop | Push (Key) |
| Inner Ethernet | Not Present | Modify (DMAC) | Ignore | Modify (DMAC) |
| Inner IP | Not Present | Ignore | Ignore | Modify (SIP,DIP) |
| TCP/UDP | Modify (SPt,DPt) | Ignore | Ignore | Modify (SPt,DPt) |

### *6.1.3 Transposition Engine*

VFP creates an action for a UFID match by composing HTs from matched rules in each layer, as in Pseudocode 1. For example, a packet passing the example Ananta NAT layer and the VL2 VNET encap layer may end up with the composite Encap+NAT transposition in Table 3. This transposition engine also contains logic to apply a transposition to an actual packet, by breaking the final transposition down into a series of steps (NAT, encap, decap) that can be applied by a packet modifier.

### *6.1.4 Unified Flow Tables and Caching*

The intuition behind our flow compiler is that the action for a UFID is relatively stable over the lifetime of a flow – so we can cache the UFID with the resulting HT from the engine. Applications like Ananta create per-connection state already, so it's not expensive to cache this whole unified flow (UF) per TCP/UDP flow. The resulting flow table where the compiler caches UFs is called the Unified Flow Table (UFT).

```
Process(UFID input, Port port):
  Transposition action = {0};
  For each layer in port.layers:
    UFID localId = Transpose(input, action);
    Rule rule = Classify(layer, localId);
    action = action.compose(rule.process(localId));
  return composite;
```

**Pseudocode 1. Transposition Engine**

With the UFT, we segment our datapath into a fastpath and a slowpath. On the first packet of a TCP flow, we take a slowpath, running the transposition engine and matching at each layer against rules. On subsequent packets, VFP takes a fastpath, matching a unified flow via UFID, and applying a transposition directly. This operation is independent of the layers or rules in VFP.

The UFT is used similarly to the OVS microflow cache to skip tables, and scales well across CPUs because it requires no write lock to match. However, a key difference for our workload is the HT, which combines encap/decap with header modification. This allows us to have a single flow for all actions rather than one before and after a tunnel interface, and is critical for offloading flows with only a single hardware table (§9.3).

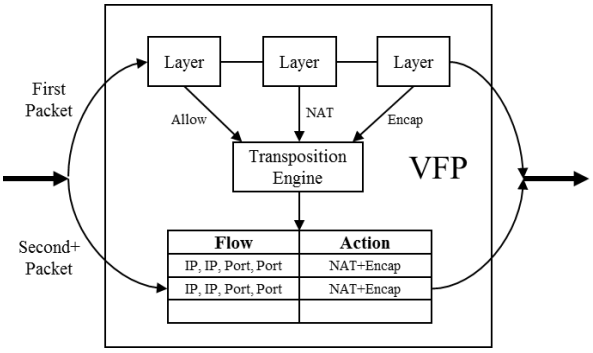


**Figure 7. VFP Unified Flow Table**

## 6.2 Action Contexts

Some rule actions have side effects beyond header modification, or take action on packet payloads. Examples include metering to a global counter (supporting our example metering layer), or encrypting packet payloads. For these actions, HTs can be extended with *Action Contexts* which can implement arbitrary logic via callback. An Action Context can be added to an HT (and the resulting UF) by a rule. This allows rules to extend the packet actions themselves even though they are not matched for every packet.

## 6.3 Flow Reconciliation

A requirement of the VFP flow compiler is transparency to VFPAPI clients. This means that if a controller changes the rules in a layer, the new rules should be

applied to subsequent packets even if a UF exists. This is supported by a reconciliation engine in VFP.

The reconciliation engine maintains a global generation number on each port. When a UF is created, it's tagged with the current generation number at creation time. Upon policy update, the port generation is incremented.

VFP implements lazy reconciliation, reconciling a UF only when matching a UF whose generation number is less than the port's current generation number. The UF is then *simulated* against the current rules on the port, by running its UFID through the transposition engine, and determining if the resulting HT has changed.

### 6.4 Flow State Tracking

By default, the expiration policy for UFs is to expire them after some configurable period of time. However, this is not efficient for short flows and leads to large numbers of UFs idling in the UFT. Instead, for TCP flows we can expire them by tracking the state of the underlying connections. This requires determining which UF should pair with a UF in the opposite direction to form a bidirectional connection.

#### *6.4.1 Flow Pairing*

Unlike layer flows, we cannot pair UFs just by reversing the FlowID – UF pairs can be asymmetric, for example if a connection is tunneled to the VM on inbound but returned directly without tunneling on the outbound side (e.g. the Direct Server Return feature in Ananta). The key idea in our solution is pairing connections on the VM-side of the VFP pipeline rather than the network side (e.g. the UFID of a connection on the inbound path after processing should be the opposite of the UFID on the outbound path before processing).

When an inbound UF is created by an inbound packet, we create an outbound UF to pair it to by reversing the UFID of the packet after the inbound action, and simulating it through the outbound path of that port, generating a full UF pair for the connection. For a new outbound UF, we wait for an inbound packet to try to create an inbound UF – when the new UF looks up the reverse UFID it will find an existing flow and pair itself.

#### *6.4.2 TCP State Tracking*

Once we have established a pairing of UFs, we use a simple TCP state machine in VFP to track them as connections. For example, new flows are created in a probationary half-open state – only when a three-way handshake is verified with proper sequence numbers does it become a full flow (this helps protect against SYN floods). We can also use this state machine to track FIN handshakes and RSTs, to expire flows early. We also track connections in TIME_WAIT, allowing NAT rules to determine when they can reuse ports safely. VFP tracks port-wide statistics such as average RTT, retransmits, and ECN marks, which can be useful in diagnosing VMs with networking issues [37].

### 6.5 Packet Classification

In practice, VFPv2's UFT datapath solved performance and scalability issues for most packets (even short lived flows are usually at least 10 packets). However, rule matching is still a factor for scenarios with thousands of rules, such as complex ACLs or routes. We implement a better classification algorithm for these cases.

We support 4 classifier types: a compressed trie, an interval tree, a hash table, and a list. Each classifier can be instantiated on each condition type from §5.3.1. We assign each rule to a classifier based on a heuristic described in detail in [40] to optimize total matching time in a VFP group. In practice we have found this to be successful at handling a wide range of rule types that users plumb, such as 5-tuple ACLs with IP ranges, and large routing tables.

## 7. Switching Model

In addition to SDN filtering, VFP also forwards traffic to ports. VFP's forwarding plane is described below.

### 7.1 Packet Forwarding

VFP implements a simple bridge, forwarding packets by destination MAC address to the port attached to the VNIC created with that MAC. This can be an outer MAC, or an inner MAC inside a VXLAN/NVGRE encaped packet for VMs running in a virtualized mode. Efficient spreading based on this policy is widely supported by NICs as Virtual Machine Queue [33].

### 7.2 Hairpinning and Mirroring

Gateway VMs are often used to bridge different tunnels or address spaces in different routing domains. To accelerate these workloads, we support hairpin rules in VFP layers, which can redirect packets to a VNIC back out through VFP ingress processing, either on the same or a different VNIC. This enables high speed gateways without the overhead of sending packets to a VM. This policy also supports programmable port mirroring.

### 7.3 QoS

VFP supports applying max cap policies on transmit and receive to ports or groups of ports on a switch. This is implemented using atomic operations to update token bucket counters and interlocked packet queues for high performance. Bandwidth reservations across ports are also supported. Our algorithm for measurement-based weighted fair sharing is described in [39].

## 8. Operational Considerations

As a production cloud service, VFP's design must take into account serviceability, monitoring and diagnostics.

### 8.1 Rebootless Updates

In an update, we first pause the datapath, then detach

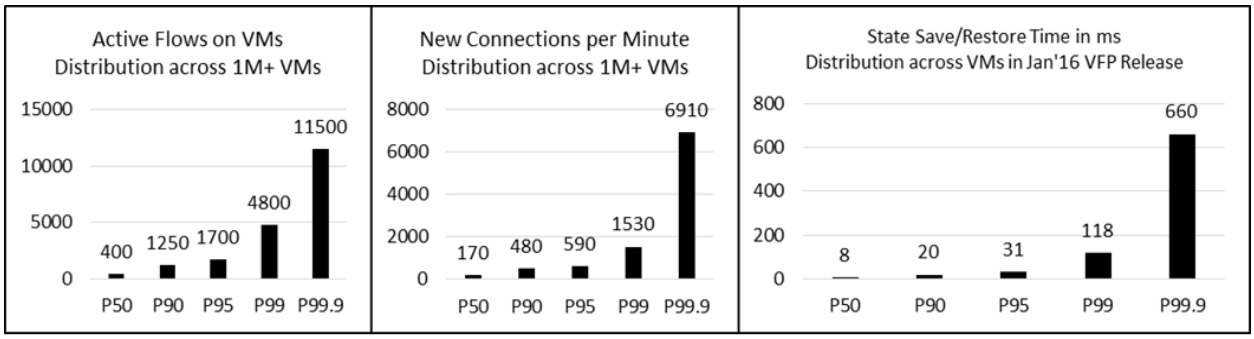


**Figure 8. Example VFP Production Monitoring Statistics**

VFP from the stack, uninstall VFP (which acts as a loadable kernel driver), install a new VFP, attach it to the stack, and restart the datapath. This operation typically completes in <1s, and looks like a brief connectivity blip to VMs, while the NIC stays up.

### 8.2 State Save/Restore

Because we support stateful policy such as ACLs and NAT, a rebootless VFP update by default forces VMs to reset all TCP connections, since flow state is lost. As we saw more and more frequent updates, we concluded we needed to build State Save/Restore (SSR) functions to eliminate impact of VFP updates to VMs.

We support serialization and deserialization for all policy and state in VFP on a port. Every VFP object has a serialization and deserialization handler, including layers/groups/rules/flows, rule contexts, action contexts, UFs, HTs, resources and resource entries, and more. All objects are versioned, so if structures are updated, SSR can support multiple source object versions.

#### *8.2.1 VM Live Migration*

VFP also supports live migration of VMs. In this case, the port state is serialized out of the original host and deserialized on the new host during the VM blackout time of the migration. VFP policies/rules are updated on the new host by all VFPAPI clients based on policy that may have changed in migration, such as the VM physical address. VFP flow reconciliation (§6.3) then handles updating flows, keeping TCP connections alive.

### 8.3 Monitoring

VFP implements over 300 performance counters and flow statistics, on per port, per layer, and per rule bases, as well as extensive flow statistics. This information is continuously uploaded to a central monitoring service, providing dashboards on which we can monitor flow utilization, drops, connection resets, and more, on a VM or aggregated on a cluster/node/VNET basis. Figure 8 shows distributions measured in production for the number of active flows and rate of new connections per VM, and SSR time for a recent VFP update.

### 8.4 Diagnostics

VFP provides diagnostics for production debugging, both of VFP itself and for VFPAPI clients. The transposition engine's simulation path can be queried on arbitrary UFIDs to provide a trace of how that UFID would behave across all rules/flows in VFP. This enables remotely debugging incorrect rules and policies.

VFP tracing, when enabled, provides detailed logs of actions performed on the data path and control path.

If we need to track a bug in VFP's logic, SSR can snapshot a port's state in production, and restore it on a local test machine, where packets can then be simulated using the above diagnostic under a kernel debugger.

## 9. Hardware Offloads and Performance

As we continue to scale up our cloud servers, we are very performance and cost sensitive. VFP implements several hardware offloads, described below.

### 9.1 Stateless Tunneling Offloads

Commercial NICs have been available since 2013 with the ability to parse NVGRE/VXLAN, and support stateless checksum, segmentation, and receive scaling offloads with them. VFP programs NICs with these offloads for tunneling, which are widely deployed in our datacenter. This has largely eliminated the performance overhead of encap/decap to our platform – with offloads we can support encap at 40Gbps line rate on our NICs.

### 9.2 QoS Offloads

Many commercial NICs support transmit max caps, and bandwdith reservations, across transmit queues. We have implemented and deployed an interface to offload port-level QoS policy from §7.3. This removes the overhead of applying QoS through software.

### 9.3 Offloading VFP Policy

Many attempts to offload SDN policy and accomplish Goal #7 of SR-IOV support focus on offloading packet classification, such that no packets traverse the host. We've found this impractical on server NICs, as it often requires large TCAMs, CPU cores or other specialized hardware when doing lookups on 10+ tables in series.

However, offloading Unified Flows as defined in §6 turns out to be much more tractable. These are exact match flows representing each connection on the system, and so they can be implemented via a large hash

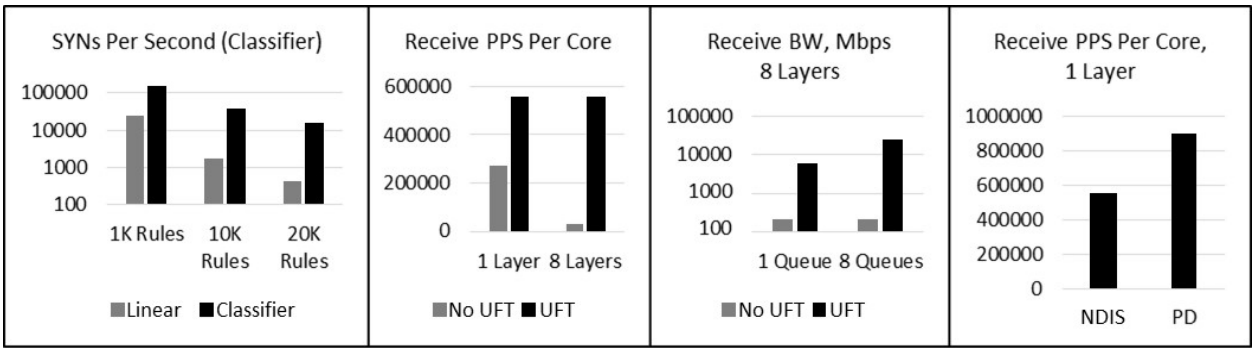


**Figure 9. Selected VFP Performance Data, 2.9Ghz Xeon Sandy Bridge CPU**

table, typically in inexpensive DRAM. In this model, the first packet of a new flow goes through software classification to determine the UF, which is then offloaded so that packets take a hardware path.

We've used this mechanism to enable SR-IOV in our datacenters with VFP policy offload on custom hardware we've deployed on all new Azure servers. Our VMs reach 25Gbps+ VNICs at line rate with near-zero host CPU and <25µs e2e TCP latencies inside a VNET.

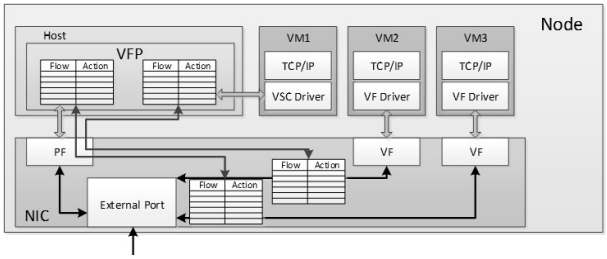


**Figure 10. VFP Offloading Flow Tables per Port**

### 9.4 Performance

In Figure 9, we first see that under a TCP SYN load the classification algorithm in §6.5 improves by 1-2 orders of magnitude over linear match a distribution of random rules. Measuring VFP's fastpath against long lived flows, we next see that on layers of 200 rules each, UFT caching improves performance even at one layer, but more dramatically as more layers are added. We then see that due to locking, in VFPv1 performance doesn't improve as we scale to 8 CPUs, while UFT scales well. Finally, we see that received packets per second improves by over 60% with PacketDirect.

## 10. Experiences

We have deployed 21 major releases of VFP since 2012. VFP runs on all Azure servers, powering millions of VMs, petabits per second of traffic, and providing load balancing for exabytes of storage, in hundreds of datacenters in over 30 regions across the world. In addition, we are releasing VFP publicly as part of Windows Server 2016 for on-premises workloads.

### 10.1 Results

We believe we accomplished all of our goals from §2:

1. We have had multiple independent controllers successfully program VFP for several years. New controllers have deployed SDN applications by inserting layers without changes in other controllers.
2. Every single connection in our datacenters is treated statefully by VFP – all pass through stateful ACLs, and many pass through a stateful NAT.
3. The definition of a VNET, a load balancer, and other policies have changed over time as newer, richer semantics were implemented. Most of these were policy changes without any change to VFP.
4. We've pushed dozens of rebootless updates to VFP.
5. The introduction of UFT dramatically improved VFP performance, especially for scenarios with large numbers of layers. This has helped us scale our servers to 40G+ NICs, with many more VMs.
6. The VFP packet classification algorithm has sped up classification on real production workloads by 1-2 orders of magnitude over linear search.
7. We have successfully offloaded VFP flows to flow-programmable hardware and deployed SR-IOV.

### 10.2 Lessons Learned

Over 5 years of developing and supporting VFP, we learned a number of other lessons of value:

- **L4 flow caching is sufficient.** We didn't find a use for multi-tiered flow caching such as OVS megaflows. The two main reasons: being entirely in the kernel allowed us to have a faster slowpath, and our use of a stateful NAT created an action for every L4 flow and so reduced the usefulness of ternary flow caching.
- **Design for statefulness from day 1.** The above point is an example of a larger lesson: support for stateful connections as a first class primitive in a MAT is fundamental and must be considered in every aspect of a MAT design. It should not be bolted on later.
- **Layering is critical.** Some of our policy could be implemented as a special case of OpenFlow tables with custom GOTOs chaining them together, with separate inbound and outbound tables. We found,

however, that our controllers needed clear layering semantics or else they couldn't reverse their policy correctly with respect to other controllers.

- **GOTO considered harmful.** Controller developers will implement policy in the simplest way needed to solve a problem, but that may not be compatible with future controllers adding policy. We needed to be vigilant in not only providing layering, but enforcing it to prevent this. We see this layering enforcement not as a limitation compared to OpenFlow's GOTO table model, but instead as the key feature that made multi-controller designs work for 4 years running.
- **IaaS cannot handle downtime.** We found that customer IaaS workloads cared deeply about uptime for each VM, not just their service as a whole. We needed to design all updates to minimize downtime, and provide guarantees for low blackout times.
- **Design for serviceability.** SSR (§8.2) is another design point that turned out to pervade all of our logic – in order to regularly update VFP without impact to VMs, we needed to consider serviceability in any new VFP feature or action type.
- **Decouple the wire protocol from the data plane.** We've seen enough controllers/agents implement wire protocols with different distributed systems models to support O(1M) scale that we believe our decision to separate VFPAPI from any wire protocol was a critical choice for VFP's success. For example, bandwidth metering rules are pushed by a controller, but VNET required a VL2-style directory system (and an agent that understands that policy comes from a different controller than pulled mappings) to scale.

  Architecturally, we believe it helps to view the resulting "Smart Agents" as part of a distributed controller application, rather than part of the dataplane, and we consider VFP's OS level API the real common Southbound API [36] in our SDN stack, where different applications meet.
- **Conflict resolution was not needed.** We believe our choice in §2.3 to not focus on automated conflict resolution between controllers was correct, as this was never really an issue when we enforced clean layering between the controllers. Good diagnostics to help controller developers understand their policy's impact were more important.
- **Everything is an action.** Modeling VL2-style encap/decap as actions rather than tunnel interfaces was a good choice. It enabled a single table lookup for all packets – no traversing a tunnel interface with tables before and after. The resulting HT language combining encap/decap with header modification enabled single-table hardware offload.
- **MTU is not a major issue.** There were initial concerns that using actions instead of tunnel interfaces would cause issues with MTU. We found this not to be a real issue – we support either making use of larger MTU on the physical network to support encapsulation (this was our choice for Azure), or using a combination of TCP Maximum Segment Size clamping and fragmentation of large non-TCP frames (for deployments without large MTU).
- **MAT Scale.** In our deployments, we typically see up to 10-20 layers, with up to hundreds of groups within a layer. We've seen up to O(50k) rules per group, (when supporting customers' distributed BGP routers). We support up to 500k simultaneous TCP connections per port (after which state tracking becomes prohibitively expensive).
- **Keep forwarding simple.** §7.1 describes our MAC-filter based forwarding plane. We considered a programmable forwarding plane based on the MAT model, however, we found no scenario for complex forwarding policy or learning. We've concluded that a programmable forwarding plane is not useful for our cloud workload, because VMs want a declarative model for creating NICs with known MAC addresses.
- **Design for E2E monitoring.** Determining network health of VMs despite not having direct access to them is a challenge. We found many uses for in-band monitoring with packet injectors and auto-responders implemented as VFP rule actions. We used these to build monitoring that traces the E2E path from the VM-host boundary. For example, we implemented Pingmesh-like [24] monitoring for VL2 VNETs.
- **Commercial NIC hardware isn't ideal for SDN.** Despite years of interest from NIC vendors about offloading SDN policy with SR-IOV, we have seen no success cases of NIC ASIC vendors supporting our policy as a traditional direct offload. Instead, large multi-core NPUs [32] are often used. We used custom FPGA-based hardware to ship SR-IOV in Azure, which we found was lower latency and more efficient.

## 11. Conclusions and Future Work

We introduced the Virtual Filtering Platform (VFP), our cloud scale vswitch for host SDN policy in Microsoft Azure. We discussed how our design achieved our dual goals of programmability and scalability. We discussed concerns around serviceability, monitoring, and diagnostics in production environments, and provided performance results, data, and lessons from real use.

Future areas of investigation include new hardware models of SDN, and extending VFP's offload language.

## Acknowledgements

We would like to thank the developers who worked on VFP since the project's inception in 2011 – Yue Zuo, Harish Kumar Chandrappa, Praveen Balasubramanian, Vikas Bhardwaj, Somesh Chaturmohta, Milan Dasgupta, Mahmoud Elhaddad, Luis Hernandez, Nathan Hu, Alan Jowett, Hadi Katebi, Fengfen Liu, Keith Mange, Randy Miller, Claire Mitchell, Sambhrama Mundkur, Chidambaram Muthu, Gaurav Poothia, Madhan Sivakumar, Ethan Song, Khoa To, Kelvin Zou, and Qasim Zuhair, as well as PMs Yu-Shun Wang, Eric Lantz, and Gabriel Silva. We thank Alireza Dabagh, Deepak Bansal, Pankaj Garg, Changhoon Kim, Hemant Kumar, Parveen Patel, Parag Sharma, Nisheeth Srivastava, Venkat Thiruvengadam, Narasimhan Venkataramaiah, Haiyong Wang, and other architects of Azure's SDN stack who had significant influence on VFP's design, as well as Jitendra Padhye, Parveen Patel, Shachar Raindel, Monia Ghobadi, George Varghese, our shepherd David Wetherall, and the anonymous reviewers, who provided valuable feedback on earlier drafts of this paper. Finally, we thank Dave Maltz, Mark Russinovich, and Albert Greenberg for their sponsorship of and support for this project over the years, and Jitendra Padhye for convincing us to write a paper about our experiences.

## References

[1] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, and P. Shelar. The Design and Implementation of Open vSwitch. In NSDI, 2015.

[2] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, S. Sengupta. VL2: A scalable and flexible data center network. In SIGCOMM, 2009.

[3] Open vSwitch – An Open Virtual Switch. http://www.openvswitch.org.

[4] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, N. Karri. Ananta: Cloud scale load balancing. In SIGCOMM, 2013.

[5] N. Mckeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner. OpenFlow: Enabling Innovation in Campus Networks. In SIGCOMM, 2008.

[6] M. Mahalingam et al. RFC 7348: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. 2015.

[7] M. Sridharan et al. RFC 7637: NVGRE: Network Virtualization Using Generic Routing Encapsulation. 2015.

[8] B. Pfaff et al. RFC 7047: The Open vSwitch Database Management Protocol. 2015.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker. P4: Programming Protocol-Independent Packet Processors. ACM Sigcomm Computer Communications Review (CCR). Volume 44, Issue #3 (July 2014).

[10] NDIS Filter Drivers. https://msdn.microsoft.com/en-us/library/windows/hardware/ff565492(v=vs.85).aspx

[11] PacketDirect Provider Interface. https://msdn.microsoft.com/en-us/library/windows/hardware/mt627758(v=vs.85).aspx.

[12] Hyper-V Extensible Switch. https://msdn.microsoft.com/en-us/library/windows/hardware/hh582268(v=vs.85).aspx

[13] Amazon Web Services. http://aws.amazon.com.

[14] Microsoft Azure. http://azure.microsoft.com.

[15] Google Cloud Platform. http://cloud.google.com.

[16] M. Challa. OpenVswitch Performance measurements & analysis. 2014. http://openvswitch.org/support/ovscon2014/18/1600-ovs_perf.pptx

[17] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, J. C. Mogul. Democratic Resolution of Resource Conflicts Between SDN Control Programs. In CoNEXT, 2014.

[18] J. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, Y. Turner. Corybantic: towards the modular composition of SDN control programs. In HotNets-XII, 2013.

[19] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In NSDI, 2013.

[20] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. Frenetic: A High-Level Language for OpenFlow Networks. In PRESTO, 2010.

[21] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In ICFP, 2011.

[22] Overview of Single Root I/O Virtualization (SR-IOV). https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-

[23] Y. Dong, X. Yang, X. Li, J. Li, K. Tan, H. Guan. High performance network virtualization with SR-IOV. In IEEE HPCA, 2010.

[24] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In SIGCOMM, 2015.

[25] A. Vahdat. Enter the Andromeda zone - Google Cloud Platform's latest networking stack. 2014. https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html

[26] Linux Bridge. Linux Foundation. https://wiki.linuxfoundation.org/networking/bridge

[27] Network Bridge Overview. Microsoft. https://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/hnw_understanding_bridge.mspx?mfr=true

[28] Hyper-V Virtual Switch Overview. Microsoft. https://technet.microsoft.com/en-us/library/hh831823(v=ws.11).aspx

[29] VTEP Schema. Open vSwitch. http://openvswitch.org/support/dist-docs/vtep.5.html

[30] vSphere 6.0 Configuration Maximums. VMWare. 2016. https://www.vmware.com/pdf/vsphere6/r60/vsphere-60-configuration-maximums.pdf

[31] Mellanox Presentation on OVS Offload. Mellanox. 2015. http://events.linuxfoundation.org/sites/events/files/slides/Mellanox%20OPNFV%20Presentation%20on%20OVS%20Offload%20Nov%2012th%202015.pdf

[32] Open vSwitch Offload and Acceleration with Agilio CX Intelligent Server Adapters. Netronome. https://www.netronome.com/media/redactor_files/WP_OVS_Benchmarking.pdf

[33] Virtual Machine Queue. Microsoft. https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/virtual-machine-queue--vmq-

[34] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, N. McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In ANCS, 2008.

[35] D. Merkel. Docker: lightweight Linux containers for consistent development and deployment. Linux Journal, Issue 239, March 2014.

[36] C. Beckmann. Southbound SDN API's. In IETF 84 SDNRG Proceedings, 2012. https://www.ietf.org/proceedings/84/slides/slides-84-sdnrg-7.pdf

[37] M. Moshref, M. Yu, R. Govindan, A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In SIGCOMM, 2016.

[38] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, J. Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In SIGCOMM, 2016.

[39] K. To, D. Firestone, G. Varghese, J. Padhye. Measurement Based Fair Queuing for Allocating Bandwidth to Virtual Machines. In HotMiddlebox, 2016.

[40] D. Firestone, H. Katebi, G. Varghese. Virtual Switch Packet Classification. In Microsoft TechReport, MSR-TR-2016-66, 2016.