FlowVisor: A Network Virtualization Layer

Rob Sherwood*, Glen Gibb†, Kok-Kiong Yap†,
Guido Appenzeller†, Martin Casado◇, Nick McKeown†, Guru Parulkar†

* Deutsche Telekom Inc. R&D Lab, † Stanford University, ◇ Nicira Networks

OPENFLOW-TR-2009-1

Abstract:

Network virtualization has long been a goal of of the network research community. With it, multiple isolated logical networks each with potentially different addressing and forwarding mechanisms can share the same physical infrastructure. Typically this is achieved by taking advantage of the flexibility of software (*e.g.* [20, 23]) or by duplicating components in (often specialized) hardware[19].

In this paper we present a new approach to switch virtualization in which the same hardware forwarding plane can be shared among multiple logical networks, each with distinct forwarding logic. We use this switch-level virtualization to build a research platform which allows multiple network experiments to run side-by-side with production traffic while still providing isolation and hardware forwarding speeds. We also show that this approach is compatible with commodity switching chipsets and does not require the use of programmable hardware such as FPGAs or network processors.

We build and deploy this virtualization platform on our own production network and demonstrate its use in practice by running five experiments simultaneously within a campus network. Further, we quantify the overhead of our approach and evaluate the completeness of the isolation between virtual slices.

October 14, 2009

Available at `http://OpenFlowSwitch.org/downloads/technicalreports/openflow-tr-2009-1-flowvisor.pdf`

FlowVisor: A Network Virtualization Layer

Rob Sherwood[*], Glen Gibb[†], Kok-Kiong Yap[†], Guido Appenzeller[†],
Martin Casado[◇], Nick McKeown[†], Guru Parulkar[†]
[*] Deutsche Telekom Inc. R&D Lab, Los Altos, CA USA
[†] Stanford University, Palo Alto, CA USA
[◇] Nicira Networks, Palo Alto, CA USA

ABSTRACT

Network virtualization has long been a goal of of the network research community. With it, multiple isolated logical networks each with potentially different addressing and forwarding mechanisms can share the same physical infrastructure. Typically this is achieved by taking advantage of the flexibility of software (*e.g.* [20, 23]) or by duplicating components in (often specialized) hardware[19].

In this paper we present a new approach to switch virtualization in which the same hardware forwarding plane can be shared among multiple logical networks, each with distinct forwarding logic. We use this switch-level virtualization to build a research platform which allows multiple network experiments to run side-by-side with production traffic while still providing isolation and hardware forwarding speeds. We also show that this approach is compatible with commodity switching chipsets and does not require the use of programmable hardware such as FPGAs or network processors.

We build and deploy this virtualization platform on our own production network and demonstrate its use in practice by running five experiments simultaneously within a campus network. Further, we quantify the overhead of our approach and evaluate the completeness of the isolation between virtual slices.

1. INTRODUCTION

This paper explores how to virtualize a network, and describes a particular system that we prototyped - called FlowVisor - that we have deployed to *slice*[1] our own production network. Similar to computer virtualization [22, 1, 21, 17], network virtualization promises to improve resource allocation, permits operators to checkpoint their network before changes, and allows competing customers to share the same equipment in a controlled and isolated fashion. Critically, virtual networks also promise to provide a safe and realistic environment to deploy and evaluate experimental "clean slate" protocols in production networks.

To better understand virtual networking, we first look closely at computer virtualization. Computer virtualization's success can be linked to a clean abstraction of the underlying hardware. That is, the computer virtualization layer has a hardware abstraction that permits slicing and sharing of resources among the guest operating systems. The effect is that each OS believes it has its own private hardware. A well defined hardware abstraction enables rapid innovation both above and below the virtualization layer. Above, the ability to build on a consistent hardware abstraction has allowed operating systems to flourish (e.g., UNIX, MacOS, several flavors of Linux, and Windows) and even encouraged entirely new approaches [24, 28]. Below, different hardware can be used (e.g., Intel, AMD, PPC, Arm, even Nvidia's GPU), so long as it can be mapped to the hardware abstraction layer. This allows different hardware to have different instruction sets optimized for higher performance, lower power, graphics, etc. Allowing choice above and below the virtualization layer means a proliferation of options, and a more competitive, innovative and efficient marketplace.

Our goal is to achieve the same benefits in the network. Thus, by analogy, the network itself should have a hardware abstraction layer. This layer should be easy to slice so that multiple wildly different networks can run simultaneously on top without interfering with each other, on a variety of different hardware, including switches, routers, access points, and so on. Above the hardware abstraction layer, we want new protocols and addressing formats to run independently in their own isolated slice of the same physical network, enabling networks optimized for the applications running on them, or customized for the operator who owns them. Below the virtualization layer, new hardware can be developed for different environments with different speed, media (wireline and wireless), power or fanout requirements.

The equipment currently deployed in our networks

[1]Borrowing from the GENI [4] literature, we call an instance of a virtual network a *slice*, and two distinct virtual networks on the same physical hardware *slices*.

was not designed for virtualization and has no common hardware abstraction layer. While individual technologies can slice particular hardware resources (e.g., MPLS can virtualize forwarding tables) and layers (e.g., WDM slices the physical layer, VLANs slices the link layer), there is currently no one single technology or clean abstraction that will virtualize the network as a whole. Further, it's not clear how—or if it's even possible—to virtualize the equipment already deployed in our networks. Commodity switches and routers typically have proprietary architectures—equipment manufacturers do not publish their full design—with limited mechanisms to change and control the equipment's software, e.g., operators can only change a limited set of configuration parameters via the command-line interface.

The specific system described in this paper builds on OpenFlow [13] as an abstraction of the underlying hardware. As we describe later, OpenFlow offers most of what we need for a hardware abstraction layer. In principle, other abstraction layers could be used, although we're not aware of any available today that meets our needs.

In this paper, we describe FlowVisor: A network virtualization layer that we built and deployed in our network. Much as a hypervisor resides between software and hardware on a PC, the FlowVisor uses OpenFlow as a hardware abstraction layer to sit logically between control and forwarding paths on a network device. To get hands-on experience in running virtual networks, we deployed FlowVisor into our own production network and use it to create experimental and production virtual slices of our campus wireline and wireless network. The resulting virtual network runs on existing or new low-cost hardware, runs at line-rate, and is backwardly compatible with our current legacy network. We have gained some experience using it as our every-day network, and we believe the same approach might be used to virtualize networks in data centers, enterprises, homes, WANs and so on.

Our goal is not to claim that our system is perfect - as we will see, there are several open questions. Rather, we are trying to understand what is easy and what is hard, and to guide the evolution of our hardware abstraction to make it easier in future to virtualize the network.

Our roadmap for the rest of the paper is as follows. We first describe our specific vision of network virtualization in § 2. In §3 we describe the FlowVisor's design and architecture, and how we implement strong isolation (§4) between virtual instances. We validate(§5) the FlowVisor's isolation capabilities and quantify its overhead. In §6 we describe the FlowVisor's deployment in our production network and our experience using it to run network experiments in distinct virtual networks on the same physical network. We finish with some concluding remarks.

2. NETWORK VIRTUALIZATION

In order to virtualize a network, we need to know what resources we are trying to slice. We argue that there are five primary slicing dimensions:

Bandwidth. It should be possible to give each slice its own fraction of bandwidth on a link. This requires a basic primitive to divide link bandwidth. There are well-known ways to do this, and the hardware abstraction can provide some choice as to how it is implemented (e.g. WDM in a optical network, emulated circuits in a packet-switched network). All elements on the forwarding path that limit the forwarding rate need to sliced too. For example, if forwarding takes place in software, then the CPU needs to be virtualized too.

Topology. Each slice should have its own view of network nodes (e.g., switches and routers) and the connectivity between them. In this way, slices can experience virtual network events such as link failure and forwarding loops.

Traffic. It should be possible to associate a specific set of traffic to one (or more) virtual networks so that one set of traffic can be cleanly isolated from another. Here, traffic is defined broadly. It could be all packets to/from a set of addresses; it could be all traffic belonging to a group of users. It could be quite specific, such as all http traffic, or all traffic with even number Ethernet addresses; or very general such as a specific user's experiment. In general we believe the hardware abstraction should work with – but not be in any way constrained by – the specific layering structures in use today. It should provide a way to slice the forwarding plane even as new protocols and address formats are defined at any layer. This suggests a very flexible way to define and slice the header space (or "flowspace" as we will call it later).

Device CPU. Switches and routers have computational resources that must be sliced. Without proper CPU slicing, switches will stop forwarding slow-path packets (*e.g.*, packets with IP options, IGMP join/leave messages), updating statistic counters (*e.g.*, SNMP) and, more importantly, will stop processing updates to the forwarding table (*e.g.*, route changes).

Forwarding Tables. Network devices typically support a finite number of forwarding rules (e.g., TCAM entries). Failure to isolate forwarding entries between slices might allow one slice to prevent another from forwarding packets.

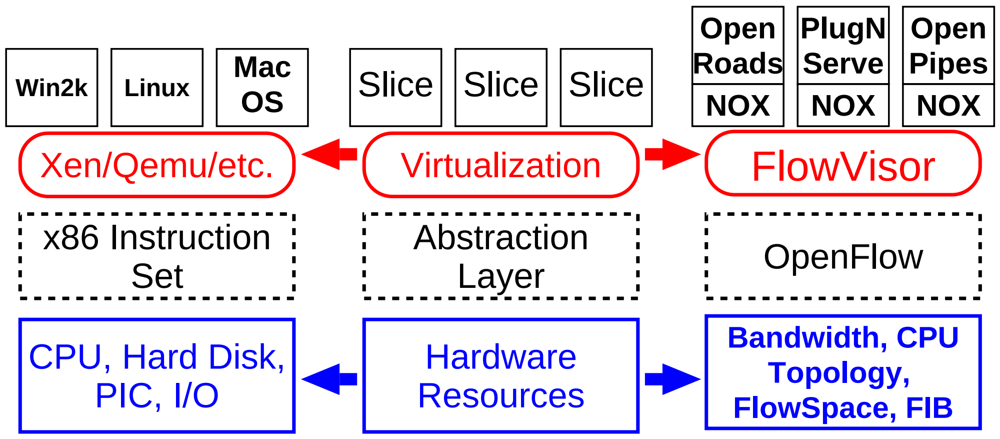

Figure 1: Similar to computer virtualization, FlowVisor is a network virtualization layer that resides between the hardware and software architectural components. OpenRoads, PlugNServe, and OpenPipes are examples of virtual network controllers built on NOX(§ 6).

3. FLOWVISOR ARCHITECTURE

Like the virtualization layer on a computer, FlowVisor sits between the underlying physical hardware and the software that controls it (Figure 1). And like an operating system uses an instruction set to control the underlying hardware, FlowVisor uses the OpenFlow protocol to control the underlying physical network. OpenFlow exposes forwarding control of a switch's packets to a programmable entity, i.e., the OpenFlow *controller*. OpenFlow is further described elsewhere [13, 15], and for the purposes of this paper a brief appendix summarizes its main characteristics (Appendix A). FlowVisor hosts multiple guest OpenFlow controllers, one controller per slice, making sure that a controller can observe and control its own slice, while isolating one slice from another (both the datapath traffic belonging to the slice, and the control of the slice).

Broadly speaking–and in a way we make concrete later–OpenFlow provides an abstraction of the networking forwarding path that allows FlowVisor to slice the network along the five required dimensions, and with the following main characteristics:

- FlowVisor defines a slice as a set of flows running on a topology of switches.[2]
- FlowVisor sits between each OpenFlow controller and the switches, to make sure that a guest controller can only observe and control the switches it is supposed to.
- FlowVisor partitions the link bandwidth by assigning a minimum data rate to the set of flows that make up a slice.
- FlowVisor partitions the flow-table in each switch by keeping track of which flow-entries belong to each guest controller.

FlowVisor is implemented as an OpenFlow proxy that intercepts messages between OpenFlow-enabled switches and OpenFlow controllers (Figure 2).

3.1 Flowspace

The set of flows that make up a slice can be thought of constituting a well-defined subspace of the entire geometric space of possible packet headers. For example, the current version of OpenFlow[3] supports forwarding rules, called *flow entries*, that match on any subset of bits in 10 fields of the packet header (from the physical port the packet arrived on, the MAC addresses, through to the TCP port numbers). The 10 fields are 256 bits long in total. If we specify a flow as a match on a specific 256 bit string, then we are defining one point (out of 2^{256}) in a 256-dimensional geometric space. Using wild cards (or bit masks), we can define any region within the space. For example, if we describe a flow with $256 - k$ '0' or '1' bits, and k wildcard or 'X' bits, then we are defining a k-dimensional region. This is a simple generalization of the commonly used geometric representation of access control lists (ACLs) for packet classification [11].

Because FlowVisor defines a slice as a set of flows, we can think of a slice as being defined by a set of (possibly non-contiguous) regions; which we call the

[2]OpenFlow can in principle be added to Ethernet switches, routers, circuit switches, access points and base stations. For brevity, we refer to any OpenFlow-enabled forwarding element as a "switch".

[3]Version 0.90 is the latest.

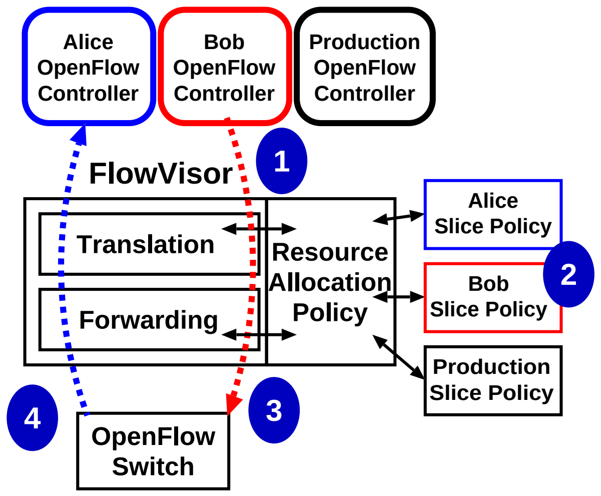

Figure 2: The FlowVisor intercepts OpenFlow messages from guest controllers (1) and, using the user's slicing policy (2), transparently rewrites (3) the message to control only a slice of the network. Messages from switches (4) are forwarded only to guests if it matches their slice policy.

slice's "flowspace". In general, we say that FlowVisor slices traffic using flowspaces. Given a packet header (a single "point"), FlowVisor can decide which flowspace contains it, and therefore which slice (or slices) it belongs to. FlowVisor can isolate two slices by making sure their flowspaces don't overlap anywhere in the topology; or it can decide which switches can be used to communicate from one slice to another. It can also allow a packet to belong to two or more slices; for example, if one slice is used to monitor other slices.

3.2 FlowVisor Design Goals

FlowVisor was designed with the following goals: (1) the virtualization should be transparent to the network controller, (2) there should be strong isolation between network slices, and (3) the slice definition policy should be rich and extensible. We discuss the rationale for each of these choices below.

Transparency. The virtualization layer should be transparent to both the network hardware and the controllers managing the virtual networks. The reasons for this are two-fold. First, an important motivation of virtual networks is the ability to prototype and debug protocols on realistic topologies. If the controller must be actively aware of the virtualization layer, it is possible to design a controller that functions in the virtual environment but not the real network. Second, it's important to decouple network virtualization technology from controller design so that they can be updated and improved independently. In our design, neither switch nor guest OpenFlow controller need be modified to interoperate with FlowVisor.

Isolation. The virtualization layer must enforce strong isolation between slices—even under adversarial conditions. The promises of virtualization break down if one slice is able to exhaust the resources of another. We describe the details of the isolation mechanisms in §4 and evaluate their effectiveness in §5.

Extensible Slice Definition. Because we have limited experience in operating virtual networks, it is important to have a slicing policy that is flexible, extensible, and modular. Much like an operating system scheduler allocates CPU resources among many processes, the slicing policy must allocate networking resources (§2) among network slices. We believe resource allocation among slices will be an active area of research. In FlowVisor, the slicing policy is implemented as a separate logical module for ease of development.

3.3 System Description

FlowVisor is a specialized OpenFlow controller. FlowVisor acts as a transparent proxy between OpenFlow-enabled network devices and multiple *guest* OpenFlow controllers (Figure 2). All OpenFlow messages, both from switch to guest and vice versa, are sent through FlowVisor. FlowVisor uses the OpenFlow protocol to communicate with both guests and switches. The guest controllers require no modification and believe they are communicating directly with the network devices.

We illustrate the FlowVisor's operation with the following simple example (Figure 2)—§6 describes more compelling use-cases. Imagine an experimenter (Bob) builds a guest controller that is an HTTP load-balancer designed to spread all HTTP traffic over a set of servers. While the controller will work on any HTTP traffic, Bob's FlowVisor policy slices the network so that he only sees traffic from one particular IP source address. His guest controller doesn't know the network has been sliced, so doesn't realize it only sees a subset of the HTTP traffic. The guest controller thinks it can control, i.e., insert flow entries for, all HTTP traffic from any source address. When Bob's controller sends a flow entry to the switches (*e.g.*, to redirect HTTP traffic to a particular server), FlowVisor intercepts it (Figure 2-1), examines Bob's slice policy (Figure 2-2), and rewrites the entry to include only traffic from the allowed source (Figure 2-3). Hence the controller is controlling only the flows it is allowed to, without knowing that the FlowVisor is slicing the network underneath. Similarly, messages that are sourced from the switch (*e.g.*, a new flow event—Figure 2-4) are only forwarded to guest controllers whose flowspace match the message.

Thus, FlowVisor enforces transparency and isolation between slices by inspecting, rewriting, and policing OpenFlow messages as they pass. Depending on the resource allocation policy, message type, destination, and content, the FlowVisor will forward a given message unchanged, translate it to a suitable message and forward, or "bounce" the message back to its sender in the form of an OpenFlow error message. For a message sent from guest controller to switch, FlowVisor ensures that the message acts only on traffic within the resources assigned to the guest. For a message in the opposite direction (switch to controller), the FlowVisor examines the message content to infer the corresponding guest(s) to which the message should be forwarded. Guest controllers only receive messages that are relevant to their network slice. Thus, from a guest controller's perspective, FlowVisor appears as a switch (or a network of switches); from a switch's perspective, FlowVisor appears as a controller.

FlowVisor does not require a 1-to-1 mapping between FlowVisor instances and physical switches. One FlowVisor instance can control multiple physical switches, and even virtualize another virtual network (Figure 3).

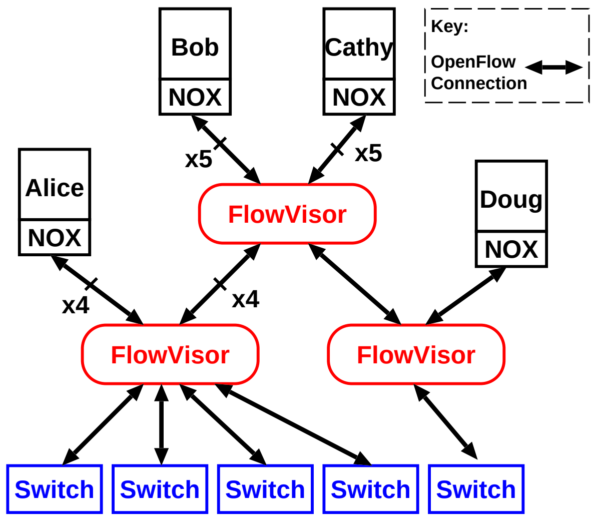

Figure 3: A single FlowVisor instance can virtualize many switches in parallel. Additionally, because of the transparent design, FlowVisor can trivially recursively slice a virtual slice, creating hierarchies of FlowVisors.

We implement FlowVisor in approximately 7000 lines of C and the code is publicly available for download.

3.4 Slice Definition Policy

Slices are defined in FlowVisor in a pluggable module. Each policy is described by a text configuration file—one per slice. For bandwidth allocation, all traffic for a slice is mapped to a single QoS class (§4.1). Each slice has a fixed, constant budget for switch CPU and forwarding table entries. Network topology is specified as a list of network nodes and ports.

The flowspace for each slice is defined by an ordered list of tuples similar to firewall rules. Each rule description has an associated action, e.g., *allow*, *read-only*, or *deny*, and is parsed in the specified order, acting on the first matching rule. Rules are combined to carve out sections of the flowspace and control delegated a particular slice. The read-only rules allow slices to receive OpenFlow control messages and query switch statistics, but not to insert entries into the forwarding table. Rules are allowed to overlap, as described in the example below.

Continuing the example from above, Alice, the network administrator, wants to allow Bob to conduct a cooperative HTTP load-balancing experiment. Bob has convinced a set of his fellow researchers to participate in his experiment. Alice wants to delegate control to Bob of just the HTTP traffic of users who have opted into the experiment, and keep the rest for her production traffic. Additionally, Alice wants to run a passive slice that monitors the performance of the whole network. This example would have the following flowspace rules.

Bob's Experimental Network is defined as the HTTP traffic for the set of users that have opted into his experiment. Thus, his network description would have one rule per user of the form:
`Allow: tcp_port:80 and ip=`*`user_ip`*. OpenFlow messages from the switch matching any of these rules are forwarded to Bob's controller. Any flow entries that Bob tries to insert are rewritten to match these rules.

Alice's Production Network is the complement of Bob's network. For each user in Bob's experiment, the production traffic network has a negative rule of the form:
`Deny: tcp_port:80 and ip=`*`user_ip`*.
The production network would have a final rule that matches all flows: `Allow: all`.
Thus, only OpenFlow messages that do not go to Bob's network are sent to the production network controller. The production controller is allowed to insert forwarding entries so long as they do not match Bob's traffic.

Alice's Monitoring Network is allowed to see all traffic in all networks. It has one rule, `Read-only: all`. This read-only rule ensures that the Alice's monitoring network is completely passive and does not interfere with the normal operation of her production network.

This rules-based policy, though simple, suffices for the experiments and deployment described in this paper. We expect that future FlowVisor deployments will have more specialized policy needs, and that researchers and virtual network operators will their own custom resource allocation policies.

4. ISOLATION MECHANISMS

A critical component of virtualization is isolation between slices. Because isolation mechanisms vary by resource, we describe each resource in turn. In addition to resources common to virtual networking(§2), our choice of OpenFlow as a hardware abstraction platform causes us to virtualize one additional resource: the OpenFlow control channel.

4.1 Bandwidth Isolation

Typically, even relatively modest commodity network hardware has some capability for basic bandwidth isolation [8]. While OpenFlow does not yet expose control of quality of service (QoS) queues, FlowVisor is still able to leverage existing switch bandwidth isolation features by marking the VLAN priority bits in packets. VLAN tags have a three bit field, the VLAN Priority Code Point (PCP), that are a standard mechanism to map a packet to one of eight priority queues. The OpenFlow protocol does expose the ability to manage VLAN tags and the priority bits, so it's possible to mark all packets in a flow with a certain priority.

Thus, to enforce bandwidth isolation, the FlowVisor rewrites all slice forwarding table additions to include a "set VLAN priority" action, setting the priority to one of eight priority queues. All traffic from a given slice is mapped to the traffic class specified by the resource allocation policy. The exact meaning of each traffic class must be configured out-of-band by the network administrator at the switch CLI.

Note that the use of VLAN PCP bits is not inherent to FlowVisor's design, but rather a short-term work-around to interoperate with commodity hardware. More direct control over QoS features and queues definitions is expected in the upcoming OpenFlow version 1.0 and will allow for more fine-grained QoS control. Also, IP's ToS bits can also be used in this same manner to map traffic to existing traffic queues.

We evaluate the effectiveness of bandwidth isolation in § 5.

4.2 Topology Isolation

Controllers discover the network's nodes and links via the OpenFlow protocol. In a non-virtual setting, a controller discover a network device when the device actively connects to the controller's listening TCP port. Since the FlowVisor acts as a proxy between switch and controller, it only proxies connections to a guest controller for the switches in the guest's virtual topology. Also, in OpenFlow there is a message to list the available physical ports on a switch. The FlowVisor simply edits the message response to report only ports that appear in the virtual topology.

Finally, special care is taken to handle link layer discovery protocol (LLDP) messages. In NOX [6], a popular OpenFlow controller development platform, LLDP messages are sent out each switch port to do neighbor discovery. When the messages are received by the neighboring switch, they do not match any forwarding rules, and are thus sent back to the controller. By noting the sending and receiving switches, NOX can infer neighbor connectivity. Since the messages have a specific, well-known form, the FlowVisor intercepts and tags the message with the sending slices ID, so that they are sent back to the correct slice when they are received again.

4.3 Switch CPU Isolation

CPU's on commodity network hardware are typically low-power embedded processors and are thus an easily over-loaded resource. The problem is that in most hardware, a highly-loaded switch CPU will result is significant network disruption. If, for example, a CPU becomes overloaded, hardware forwarding will continue, but switches will stop responding to OpenFlow requests, which causes the LLDP neighbor discovery protocol to timeout, which causes the controller to believe there is network-wide link flapping, and then for all intents and purposes the network become unusable.

There are four main sources of load on a switch CPU: (1) generating new flow setup messages, (2) handing requests from controller, (3) forwarding "slow path" packets, and (4) internal state keeping. Each of these sources of load requires a different isolation mechanism.

New Flow Messages. In OpenFlow, when a packet arrives at a switch that does not match an entry in the flow table, a new flow message is sent to the controller. This process consumes processing resources on a switch and if message generation occurs too frequently, the CPU resources can be exhausted. To prevent starvation, the FlowVisor tracks the new flow message arrival rate for each slice, and if it exceeds some threshold, the FlowVisor will insert a forwarding rule to drop the offending packets for a short period. Thus, FlowVisor uses the OpenFlow protocol to rate limit the incoming new flow messages. We discuss the effectiveness of this technique in §5.2.3.

Controller Requests. The requests an OpenFlow controller sends to the switch, e.g., to edit the forwarding table or query statistics, consume CPU resources. For each slice, the FlowVisor limits CPU consumption by throttling the OpenFlow message rate to a maximum rate per second. Because the amount of CPU resources consumed vary by message type and by hardware implementation, it is future work to dynamically infer the cost of each OpenFlow message for each hardware platform.

Slow-Path Forwarding. Packets that traverse the "slow" path—i.e., not the "fast" dedicated hardware forwarding path—consume CPU resources. Thus, an OpenFlow rule that forwards packets via the slow path can consume arbitrary CPU resources. FlowVisor prevents guest controllers from inserting slow-path forwarding rules by rewriting them as one-time packet forwarding events, i.e., an OpenFlow "packet out" message. As a result, the slow-path packets are rate limited by the above two isolation mechanisms: new flow messages and controller request rate limiting.

Internal Bookkeeping. All network devices use CPU to update their internal counters, process events, update counters, etc. So, care must be taken to ensure that there are sufficient CPU available for the switch's bookkeeping. The FlowVisor accounts for this by ensuring that the above rate limits are tuned to leave sufficient CPU resources for the switches internal function.

As with bandwidth isolation, these CPU-isolation mechanisms are not inherent to FlowVisor's design, but rather a work-around to deal with the existing hardware abstraction. A better long-term solution would be to expose the switch's existing process scheduling and rate-limiting features via the hardware abstraction. Some architectures, e.g., the HP ProCurve 5400, already use these rate-limiters to implement switch CPU isolation between the OpenFlow and non-OpenFlow enabled VLANs. Adding these features to OpenFlow is an active point of future work.

4.4 FlowSpace Isolation

Each slices must be restricted to only affecting flows in their flowspace. The FlowVisor performs message rewriting to transparently ensure that a slice only has control over its own flows and cannot affect other slices flows. Not all rules can be rewritten to fit to a slice: the FlowVisor will only make rules more specific. So, using the previous slicing example (§3.4), if Bob's controller tried to create a rule affecting all traffic, the FlowVisor would rewrite the rule to only affect TCP traffic to port 80. However, the FlowVisor will not, for example, rewrite a rule that affects port 22 traffic to only affect port 80 traffic. In the case of rules that cannot be rewritten, the FlowVisor sends an error message back to the controller indicating that the flow entry cannot be added.

4.5 Flow Entries Isolation

The FlowVisor counts the number of flow entries used per slice and ensures that each slice does not exceed a preset limit. The FlowVisor increments a counter for each rule a guest controller inserts into the switch and then decrements the counter when a rule expires. Due to hardware limitations, certain switches will internally expand rules that match multiple input ports, so the FlowVisor needs to handle this case specially. The OpenFlow protocol also provides a mechanism for the FlowVisor to explicitly list the flow entries in a switch. When a guest controller exceeds its flow entry limit, any new rule insertions received a "table full" error message.

4.6 OpenFlow Control Isolation

In addition to physical resources, the OpenFlow control channel itself must be virtualized and isolated. For example, all messages in OpenFlow include a unique transaction identifier; the FlowVisor must rewrite the transaction IDs to ensure that messages from different guest controllers do not use the same ID. Similarly, OpenFlow uses a 32-bit integer to identify the buffer where a packet is queued while its forwarding decision is pushed up to the controller. The FlowVisor needs to ensure that each guest controller can only access its own buffers. Status messages, e.g., link-down on a port, have to be duplicated to all of the affected slices. Vitalizing the control channel is made easier because the OpenFlow protocol only defines 16 message types.

5. EVALUATION

To motivate the efficiency and robustness of the design, in this section we evaluate both the FlowVisor's performance and isolation properties.

5.1 Performance Overhead

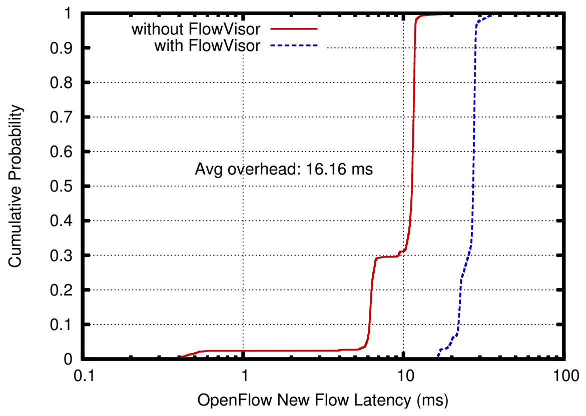

Figure 4: CDF of virtualization overhead for OpenFlow new flow messages.

Adding an additional layer between control and data paths adds overhead to the system. However, as a re-

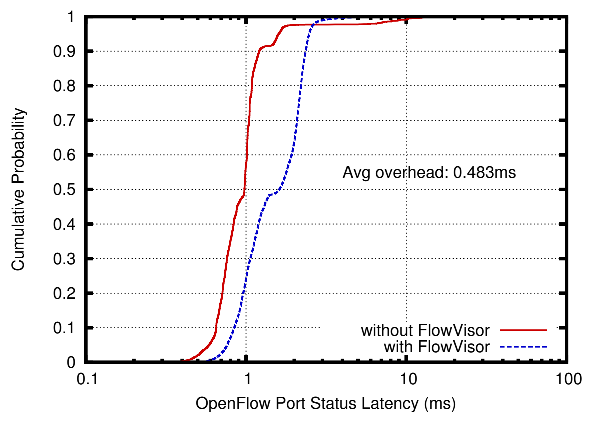

Figure 5: CDF of virtualization overhead for OpenFlow port status requests.

sult of our design, the FlowVisor does not add overhead to the data path. That is, with FlowVisor, packets are forwarded at full line rate. Nor does the FlowVisor add overhead to the control plane: control-level calculations like route selection proceed at their unvirtualized rate. FlowVisor only adds overhead to actions that cross between the control and data path layers.

To quantify this cross-layer overhead, we measure the increased response time for guest controller requests with and without the FlowVisor. Specifically, we consider the response time of the OpenFlow messages most commonly used in our network and by our monitoring software [25]: the new flow and the port status request messages.

In OpenFlow, a switch sends the controller a new flow message when a packet arrives that does not match any existing forwarding rules. We examine the overhead of the new flow message to better understand how the FlowVisor increases connection setup latency. In our experiment, we connect a machine with two interfaces to a switch. One interface sends 51 packets per second to the switch and the other interface is the OpenFlow control channel. We measure the time between sending the packet and receiving the new flow message using libpcap. Our results (Figure 4) show that the FlowVisor increases new flow message latency, i.e., the additional time from switch to controller, by 16ms on average. For latency sensitive applications, e.g., web services in large data centers, 16ms may be too much overhead. However, new flow messages add 12ms latency on average even without the FlowVisor, so we believe that guest controllers in those environments will likely proactively insert flow entries into switches, avoiding this latency all together. Additionally, the algorithm the FlowVisor uses to process new flow messages is naive, and runs in time relative to the number of flowspace rules. Our future work is to optimize this lookup process using a trie [10].

A port status request is a message sent by the controller to the switch to query the byte and packet counters for a specific port. The switch returns the counters in a corresponding port status reply message. We choose to study the ports status request because we believe it to be a worst case for FlowVisor overhead: the message is very cheap to process at the switch and controller, and the FlowVisor has to edit the message per slice to remove statistics for ports that do not appear in a slices virtual topology.

We wrote a special-purpose controller that sent port status requests at approximately 200 requests per second and measured the response times. The rate was chosen to approximate the maximum request rate supported by the hardware. The controller, switch, and FlowVisor were all on the same local area network, but controller and FlowVisor were hosted on separate PCs. Obviously, the overhead can be increased by moving the FlowVisor arbitrarily far away from the controller, but we design this experiment to quantify the FlowVisor's processing overhead. Our results show that adding the FlowVisor causes an average overhead for port status responses of 0.48 milliseconds(Figure 5). We believe that port status response time being faster than new flow processing time is not inherent, but simply a matter of better optimization for port status requests handling.

5.2 Isolation

5.2.1 *Bandwidth*

To validate the FlowVisor's bandwidth isolation properties, we run an experiment where two slices compete for bandwidth on a shared link. We consider the worst case for bandwidth isolation: the first slice sends TCP-friendly traffic and the other slice sends constant-bit-rate (CBR) traffic at full link speed (1Gbps). We believe these traffic patterns are representative of a scenario where production slice (TCP) shares a link with, for example, a slice running a DDoS experiment (CBR).

This experiment uses 3 machines—two sources and a common sink—all connected via the same HP ProCurve 5400 switch, i.e., the switch found in our wiring closet. The traffic is generated by iperf [9] in TCP mode for the TCP traffic and UDP mode at 1Gbps for the CBR traffic. We repeat the experiment twice: with and without the FlowVisor's bandwidth isolation features enabled (Figure 6). With the bandwidth isolation disabled ("without QoS"), the CBR traffic consumes nearly all the bandwidth and the TCP traffic averages 1.2% of the link bandwidth. With the traffic isolation features enabled, the FlowVisor maps the TCP slice to a QoS class that guarantees at least 70% of link bandwidth and maps the CBR slice to a class that guarantees at least 30%. Note that theses are *minimum* bandwidth guarantees, not maximum. With the bandwidth isolation features enabled, the TCP slice achieves an average of 64.2% of the total bandwidth and the CBR an average of 28.5%. Note that the event at 20 seconds where the CBR with QoS jumps and the TCP with QoS experiences a corresponding dip . We believe this to be the result of a TCP congestion event that allowed the CBR traffic to temporarily take advantage of additional available bandwidth, exactly as the minimum bandwidth QoS class is designed.

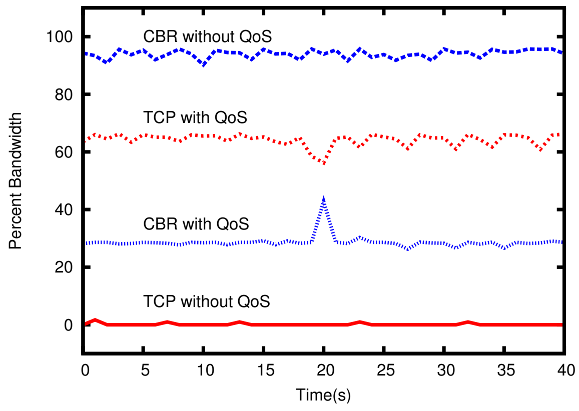

Figure 6: Effect of FlowVisor bandwidth isolation on competing TCP and CBR traffic

5.2.2 *FlowSpace*

To validate the correctness of the FlowSpace isolation algorithm, we design 21 distinct experiments. These experiments verify that slices cannot affect traffic that is not their own, that their flow entries are correctly rewritten to affect only their traffic, and that flow entry expiration messages only go to the correct slices. These test cases have been incorporated into the FlowVisor's automated testing suite.

5.2.3 *Switch CPU*

To quantify our ability to isolate the switch CPU resource, we show two experiments that monitor CPU-usage overtime of a switch with and without isolation enabled. In the first experiment (Figure 7), the OpenFlow controller maliciously sends port stats request messages (as above) at increasing speeds (2,4,8,. . .,1024 requests per second). In our second experiment (Figure 8), the switch generates new flow messages faster than its CPU can handle and a faulty controller does not add a new rule to match them. In both experiments, we show the 1-second-average switch's CPU utilization over time, and the FlowVisor's isolation features reduce the switch utilization from 100% to a configurable amount. In the first experiment, we note that the switch could handle less than 256 port status requests without appreciable CPU load, but immediately goes to 100% load when the request rate hits 256 requests per second. In the second experiment, the bursts of CPU activity in Figure 8 is a direct result of using null forwarding rules (§4.3) to rate limit incoming new flow messages. We expect that future versions of OpenFlow will better expose the hardware CPU limiting features already in switches today.

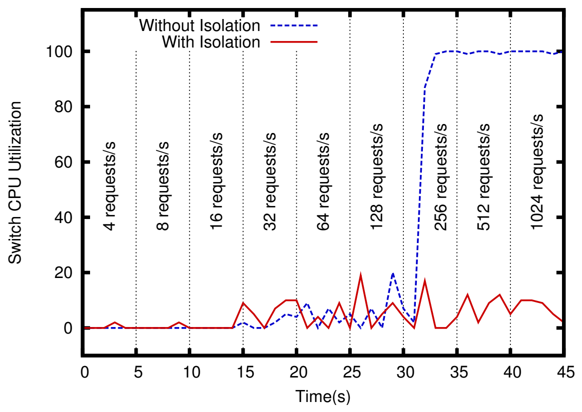

Figure 7: FlowVisor's message throttling isolation prevents a malicious controller from saturating switch CPU.

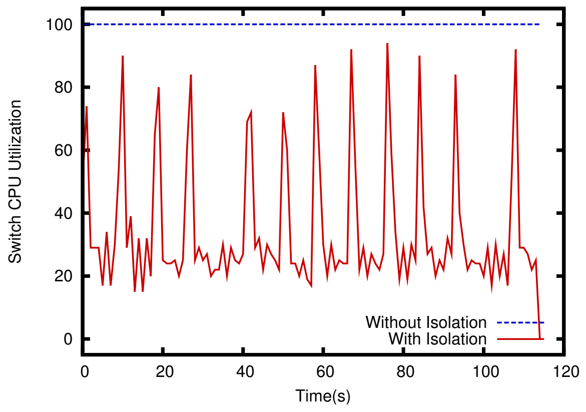

Figure 8: FlowVisor's new flow messages throttling prevents a faulty controller from saturating switch CPU.

6. DEPLOYMENT EXPERIENCE

To gain experience with running, administering, and debugging virtual networks, we deploy FlowVisor on our production networks. By "production" network, we mean the network that the authors used daily to read their email, surf the web, etc. Upon this same network, we in parallel ran four distinct network experiments—each in their own slice. In this section, we describe our experiences in deploying FlowVisor, how each experiment stressed the FlowVisor, and some preliminary results to quantity the FlowVisor's scalability.

6.1 Deployment Description

We have been running FlowVisor continuously on our production network since June 4th, 2009. Our network consists of 20 users, five NEC IP8800 switches, two HP ProCurve 5400s, 30 wireless access points, five NetFPGA [12] cards acting as OpenFlow switches, and a WiMAX base station. All network devices are pointed to a single FlowVisor instance, running on a 3.0GHz quad-core Intel Xeon with 2 GB of ram. For maximum uptime, we ran FlowVisor from a wrapper script that instantly restarts it if it should crash. The FlowVisor was able to handle restarts seamlessly because it does not maintain any hard state in the network. In our production slice, we ran NOX's routing model to perform MAC address learning and basic forwarding in the network. For future work, we hope to continue to develop and publish our virtual network tools and debugging techniques.

6.2 Experiments

On the same network, we also ran four networking experiments in parallel on our network [18]. All four experiments were built on top of NOX [6].

OpenPipes demonstrates [5] how hardware designs can be partitions over a physical network. In this experiment, the traffic consisted of video frames encapsulated in raw ethernet and was piped through various video filters running on nodes distributed through the network. OpenPipe's traffic stressed the FlowVisor flexible FlowSpace slicing in terms of its ability to slice by ethernet type.

OpenRoads experiments [27, 26] with loss-less handover between the WiMAX and wifi wireless nodes. By dynamically re-directing how traffic flows through the network, OpenRoads is able to provide finer-grained control of mobility policies, e.g., make-before-break or break-before-make connections. OpenRoads made heavy use of "read-only" FlowSpace, testing the FlowVisor's traffic isolation capabilities.

Aggregation demonstrates OpenFlow's ability to define flows as groups of TCP sessions. In the experiment, hundreds of TCP flows are "bundled" into a single flow table entry. The aggregation experiment's slice definition had 512 rules, testing the FlowVisor's processing and rewriting capabilities.

Plug-N-Serve tested [7] various algorithms for load-balancing web requests in unstructured networks. In this experiment, web queries arrive at a configurable rate and are load-balanced both by path and by server. The Plug-N-Serve experiment's query generator tested FlowVisor's new flow switch CPU isolation features.

6.3 Preliminary Scaling Evaluation

FlowVisor trivially scales to the size of our current deployment. In terms of computation, the FlowVisor uses at most 5% CPU on a 3.0GHz Xeon. In terms of bandwidth, our busiest switch in the middle of the day (i.e., the busiest time) averaged 8.6 KB/s of control traffic with a peak of 137KB/s (0.01% of a 1 Gbps link), as measured by the sum of control traffic over all slices over 3 hours.

A larger deployment is required to properly evaluate the scaling limits of a single FlowVisor instance. That said, if we assume that the FlowVisor's CPU and bandwidth consumption scale linearly with the number of controllers and input the above peak-utilization performance numbers, it is possible infer the scaling limits. Using the 5% peak CPU-utilization measurement, the number of slices could grow by a factor of 20 before the FlowVisor became CPU bound. The current version of FlowVisor runs in a single thread, so a multi-threaded FlowVisor would allow even further scaling. Assuming 137KB/s of control traffic per switch across 5 controllers (four experimental controllers and one production controller), then the FlowVisor could in theory handle on the order of thousands of controllers concurrently before becoming bandwidth bottle-necked. Adding more and faster interfaces would further increase the FlowVisor's bandwidth scaling.

7. RELATED WORK

VLANs [2] are widely used for segmentation and isolation in networks today. VLANs virtualize standard Ethernet L2 broadcast domains by decoupling the virtual links from the physical ports. This allows multiple virtual links to be multiplexed over a single virtual port (*trunk mode*), and it allows a single switch to be segmented into multiple, L2 broadcast networks. However, VLANs differ from FlowVisor in that rather than virtualizing the network control layer generally, they virtualize a specific forwarding algorithm (L2 learning). FlowVisor, on the other hand, not only supports a much more flexible method of defining networks over flow space, it provides a model for virtualizing any forwarding logic which conforms to the basic flow model.

Orphal [14] is, a low-level API for implementing forwarding and control logic within a production switch. Significantly, Orphal provides isolation between distinct forwarding applications (switchlets) within the same switch. Orphal differs from our work in two ways. First, it defines an interface to a single switch rather than a network of switches. In fact, because Orphal has been used to implement OpenFlow, it could participate on a FlowVisor managed network. Second, Orphal was design to provide low-level access to programmable hardware such as on-board ASICs. In contrast, FlowVisor maintains the OpenFlow interface of providing a flow-level interface to the control software.

There is a vast array of work related to network experimentation in both controlled and operational environments. Here we merely scratch the surface by discussing some of the more recent highlights.

The community has benefited from a number of testbeds for performing large-scale network experiments. Two of the most widely used are PlanetLab [16] and EmuLab [23]. PlanetLab's primary function has been that of an overlay testbed, hosting software services on nodes deployed around the globe. EmuLab is targeted more at localized and controlled experiments run from arbitrary switch-level topologies connected by PCs. VINI [20], a testbed closely affiliated with PlanetLab, further provides the ability for multiple researchers to construct arbitrary topologies of software routers while sharing the same physical infrastructure. Similarly, software virtual routers offer both programmability, reconfigurability, and have been shown to manage impressive throughput on commodity hardware (e.g., [3]).

In the spirit of these and other testbed technologies, FlowVisor is designed to aid research by allowing multiple projects to operate simultaneously, and in isolation, in realistic network environments. What distinguishes our approach is that our focus is on providing segmentation and isolation of the hardware forwarding paths of real commercial networking gear. We also explore how to run experiments while sharing switch-level infrastructure of operational networks.

Supercharging PlanetLab [19] and GENI [4] propose network experimentation platforms designed around network processors. These have the advantage of both providing extremely high performance and isolation while allowing for sophisticated per-packet processing. In contrast, our work concedes the ability to perform arbitrary per-packet computation in order achieve broad cross-vendor support. We believe our work to be complimentary and well-suited for integration with network processor-based platforms.

8. FUTURE WORK

Our experience in deploying FlowVisor has exposed many unanswered questions. Optimal, general, or even practical solutions to virtual network resource allocation, management, and debugging have potential to be fruitful areas of research. More concretely, the current FlowVisor implementation only virtualizes switch forwarding logic, the traffic's flowspace, and associated hardware resources (*e.g.*, bandwidth/queues, topology/ports). However, there are other resources on a network that could also be virtualized. Here we outline three such resources and describe how they might be virtualized.

Virtual Device Configuration. FlowVisor does not allow slices to affect the configuration of switches. That is, users cannot change or, for example, enable a low-power transmission mode. We expect device configuration to be a critical feature in virtualizing wireless networks, where users may want to set transmission power, change the advertised ESSID, etc. We are currently drafting a simple device configuration protocol for the OpenFlow protocol, with the intent of then virtualizing it.

Virtual Links. FlowVisor's virtual topologies are restricted to subsets of the physical topology. The ability to create complex virtual topologies completely decouple from the underlying physical connectivity is likely of use to clean slate Internet design. We expect that support for tunneling will be added to the OpenFlow protocol in the near future.

Virtual Address Space. Currently, FlowVisor has no way for two slices to share flowspace and simultaneously prevent them from interfering with each other's traffic. This is potentially problematic, as two slices might want control over the same desirable flowspace, e.g., the 10.0.0.0/8 IP netblock. Similar to virtual memory, it is possible to use packet rewriting to *virtually* allocate each slice the same flowspace, and transparently rewrite it to non-overlapping regions.

9. CONCLUSION

We originally set out to answer a specific problem: How can we slice our campus production network so that every student in a projects class can each run their own network experiment to control forwarding decisions, all at the same time? We concluded early on that if we want to slice a network built from different types of network device (Ethernet switches, routers, APs, etc), then we need a way to abstract the forwarding plane that is common across all network devices, and is not specific or confined to one protocol layer. In order to slice the forwarding plane, we concluded that we need: (1) For each packet, a way to determine and enforce which slice (or slices) it belongs to, (2) To isolate the bandwidth of one slice from another, and (3) To allow the forwarding decisions for each slice to be controlled independently by its own control logic.

There are several ways to determine and enforce which slice a packet belongs to: For example, any commonly agreed upon header field could be used to multiplex and de-multiplex packets into the set of flows that make up a slice. But to give us most flexibility in the definition of a flow (and hence a slice), we wanted flow definitions that include legacy packet formats (for backward compatibility), can work across multiple protocol layers (to allow for experiments at different layers, and to allow for different definitions of a flow in different parts of the network - for example bigger flows on aggregated links), and allow the protocol to be changed in any layer.

We came to the conclusion that *flow space* is a critical property when virtualizing a network. The more freedom the forwarding plane gives us to express flow space - with as many headers and as few constraints as possible - then the richer the slices we can create, and the easier it is to define and use new protocols. We have already seen much richer slice descriptions in FlowVisor than we expected; slice definitions have included multiple new header fields, the time of day, and the identity of the user. It is hard to imagine any other way to accomplish the same expressibility.

We used OpenFlow in our prototype because it comes closest to meeting our requirements. However, in its current version, OpenFlow matches on specific headers (L2-L4). In future versions, motivated in part by our work on slicing, we expect OpenFlow will match on many more bits in packet headers, as well as allow control of layer-1 transport networks.

There are several ways to isolate the bandwidth of one slice from another. FlowVisor does this by assigning bandwidth to the flows in each slice. This prompted us to extend OpenFlow so that a flow (or set of flows) can be assigned a minimum data rate.

We have deployed FlowVisor in our production network at Stanford, and we are currently extending it to three buildings in the school of engineering. In 2010 we will work with researchers and IT staff on seven US college campuses - as well as the Internet2 and NLR backbones - to help deploy FlowVisor in their networks too.

10. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[2] L. S. Committee. Ieee802.1q - ieee standard for local and metropolitan area networksvirtual bridged local area networks. IEEE Computer Society, 2005.

[3] N. Egi, M. Hoerdt, L. Mathy, F. H. Adam Greenhalgh, and M. Handley. Towards High Performance Virtual Routers on Commodity Hardware. In *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2008.

[4] GENI.net Global Environment for Network Innovations. `http://www.geni.net`.

[5] G. Gibb, D. Underhill, A. Covington, T. Yabe, and N. McKeown. OpenPipes: Prototyping high-speed networking systems. In *"Proc. ACM SIGCOMM Conference (Demo)"*, Barcelona, Spain, August 2009.

[6] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards and operating system for networks. In *ACM SIGCOMM Computer Communication Review*, July 2008.

[7] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow. In *ACM SIGCOMM Demo*, August 2009.

[8] Advanced traffic management guide. HP ProCurve Switch Software Manual, March 2009. `www.procurve.com`.

[9] iperf - network performance tool. `iperf.sourceforge.net`.

[10] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition.*, chapter Chapter 6.3, page 492. Addison Wesley, 1997.

[11] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 203–214, New York, NY, USA, 1998. ACM.

[12] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, 2007.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer*

Communication Review, 38(2):69–74, April 2008.

[14] J. C. Mogul, P. Yalagandula, J. Tourrilhes, R. McGeer, S. Banerjee, T. Connors, and P. Sharma. Orphal: Api design challenges for open router platforms on proprietary hardware. Technical Report 108, HP Laboratories, 2008.

[15] The OpenFlow Switch Consortium. `http://www.openflowswitch.org`.

[16] An open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org/`.

[17] `www.qemu.org`.

[18] R. Sherwood, M. Chan, G. Gibb, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, D. Underhill, K.-K. Yap, and N. McKeown. Carving research slices out of your production network with openflow. In *Sigcomm 2009 Demo Session*. ACM Sigcomm, August 2009.

[19] J. S. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, C. Wiseman, and D. Zar. Supercharging planetlab: a high performance, multi-application, overlay network platform. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 85–96, New York, NY, USA, 2007. ACM.

[20] A virtual network infrastructure. `http://www.vini-veritas.net`.

[21] `www.virtualbox.org`.

[22] VMware. `http://www.vmware.com`.

[23] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[24] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2008.

[25] K. Yap, M. Kobayashi, D. Underhill, S. Seetharamam, P. Kazemian, and N. McKeown. The stanford openroads deployment. In *ACM Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization (WiNTECH)*, September 2009.

[26] K.-K. Yap, T.-Y. Huang, M. Kobayashi, M. Chan, R. Sherwood, G. Parulkar, and N. McKeown. Lossless Handover with n-casting between WiFi-WiMAX on OpenRoads. In *ACM Mobicom (Demo)*, 2009.

[27] K.-K. Yap, M. Kobayashi, R. Sherwood, N. Handigol, T.-Y. Huang, M. Chan, and N. McKeown. OpenRoads: Empowering research in mobile networks. In *Proceedings of ACM SIGCOMM (Poster)*, Barcelona, Spain, August 2009.

[28] N. Zeldovich, S. Boyd-wickizer, E. Kohler, and D. Mazires. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 263–278. USENIX Association, 2006.

APPENDIX

A. BRIEF OPENFLOW DESCRIPTION

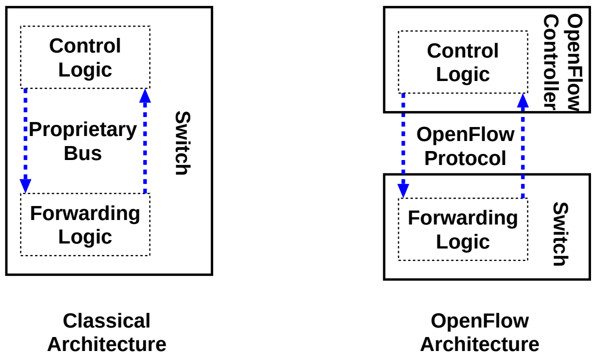

Figure 9: Relative to traditional hardware, the OpenFlow protocol moves the control path to an external controller.

A thin and efficient virtualization layer requires a clean hardware interface. In this capacity, FlowVisor is built on top of OpenFlow. While OpenFlow was originally designed as an open interface for controlling how packets are forwarded, it has the secondary effect of abstracting the underlying hardware. In this Appendix, we briefly describe OpenFlow's functionality. It is worth noting that the FlowVisor's architecture is not inheriently tied to OpenFlow: FlowVisor could in theory leverage another hardware abstraction were it available. However, to the best of our knowledge, OpenFlow is the only cross-vendor hardware abstraction protocol available today.

OpenFlow [13, 15] is an open standard that allows researchers to directly control the way packets are routed in the network. In a classical network architecture, the control logic and the data path are co-located on the same device and communicate via an internal proprietary bus. In OpenFlow, the control logic is moved to an external controller (typically a commodity PC); the controller talks to the datapath (over the network itself) using the OpenFlow protocol (Figure 9). The OpenFlow protocol abstracts forwarding/routing directives as "flow entries". A flow entry consists of a bit pattern, a list of actions, and a set of counters. Each flow entry states "perform this list of actions on all packets in this flow" where a typical action is "forward the packet out port X" and the flow is defined as the set of packets that match the given bit pattern. The collection of flow entries on a network device is called the "flow table".

When a packet arrives at a switch or router, the device looks up the packet in the flow table and performs the corresponding set of actions. When a packet does not have an entry in the flow table (i.e., it does not match any existing bit patterns) the packet is queued and a new flow event is sent to the external controller. The controller responds with a flow modification message which adds a new rule to the flow table to handle the queued packet. The new flow entry is cached in the flow table and subsequent packets in the same flow will be handled by it. Thus, the external controller is only contacted for the first packet in a flow and the subsequent packets are forwards at the switch's full line rate.

Architecturally, OpenFlow exploits the fact that modern switches and routers already logically implement flow entries and flow tables—typically in hardware as TCAMs. As such, a network device can be made OpenFlow-compliant via firmware upgrade, i.e., without additional hardware support.