Can the Production Network Be the Testbed?

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

[*] *Deutsche Telekom Inc. R&D Lab, Los Altos, CA*[†] *Stanford University, Palo Alto, CA*
[◇] *Nicira Networks, Palo Alto, CA* [‡] *Big Switch Networks, Palo Alto, CA*

Abstract

A persistent problem in computer network research is validation. When deciding how to evaluate a new feature or bug fix, a researcher or operator must trade-off realism (in terms of scale, actual user traffic, real equipment) and cost (larger scale costs more money, real user traffic likely requires downtime, and real equipment requires vendor adoption which can take years). Building a realistic testbed is hard because "real" networking takes place on closed, commercial switches and routers with special purpose hardware. But if we build our testbed from software switches, they run several orders of magnitude slower. Even if we build a realistic network testbed, it is hard to scale, because it is special purpose and is in addition to the regular network. It needs its own location, support and dedicated links. For a testbed to have global reach takes investment beyond the reach of most researchers.

In this paper, we describe a way to build a testbed that is embedded in—and thus grows with—the network. The technique—embodied in our first prototype, FlowVisor—slices the network hardware by placing a layer between the control plane and the data plane. We demonstrate that FlowVisor slices our own production network, with legacy protocols running in their own protected slice, alongside experiments created by researchers. The basic idea is that if unmodified hardware supports some basic primitives (in our prototype, OpenFlow, but others are possible), then a worldwide testbed can ride on the coat-tails of deployments, at no extra expense. Further, we evaluate the performance impact and describe how FlowVisor is deployed at seven other campuses as part of a wider evaluation platform.

1 Introduction

For many years the networking research community has grappled with how best to evaluate new research ideas.

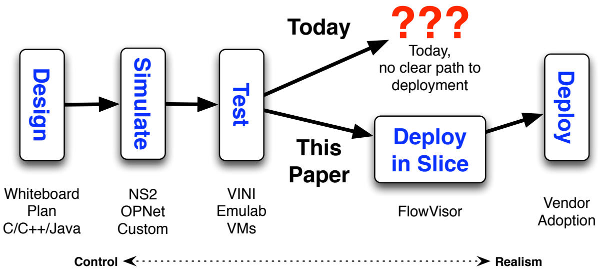

Figure 1: Today's evaluation process is a continuum from controlled but synthetic to uncontrolled but realistic testing, with no clear path to vendor adoption.

Simulation [17, 19] and emulation [25] provide tightly controlled environments to run repeatable experiments, but lack scale and realism; they neither extend all the way to the end-user nor carry real user traffic. Special isolated testbeds [10, 22, 3] allow testing at scale, and can carry real user traffic, but are usually dedicated to a particular type of experiment and are beyond the budget of most researchers.

Without the means to realistically test a new idea there has been relatively little technology transfer from the research lab to real-world networks. Network vendors are understandably reluctant to incorporate new features before they have been thoroughly tested at scale, in realistic conditions with real user traffic. This slows the pace of innovation, and many good ideas never see the light of day.

Peeking over the wall to the distributed systems community, things are much better. PlanetLab has proved invaluable as a way to test new distributed applications at scale (over 1,000 nodes worldwide), realistically (it runs real services, and real users opt in), and offers a straightforward path to real deployment (services developed in a PlanetLab slice are easily ported to dedicated servers).

In the past few years, the networking research community has sought an equivalent platform, funded by programs such as GENI [8], FIRE [6], etc. The goal is to allow new network algorithms, features, protocols or services to be deployed at scale, with real user traffic, on a real topology, at line-rate, with real users; and in a manner that the prototype service can easily be transferred to run in a production network. Examples of experimental new services might include a new routing protocol, a network load-balancer, novel methods for data center routing, access control, novel hand-off schemes for mobile users or mobile virtual machines, network energy managers, and so on.

The network testbeds that come closest to achieving this today are VINI [1] and Emulab [25]: both provide a shared physical infrastructure allowing multiple simultaneous experiments to evaluate new services on a physical testbed. Users may develop code to modify both the data plane and the control plane within their own isolated topology. Experiments may run real routing software, and expose their experiments to real network events. Emulab is concentrated in one location, whereas VINI is spread out across a wide area network.

VINI and Emulab trade off realism for flexibility in three main ways.

Speed: In both testbeds packet processing and forwarding is done in software by a conventional CPU. This makes it easy to program a new service, but means it runs much slower than in a real network. Real networks in enterprises, data centers, college campuses and backbones are built from switches and routers based on ASICs. ASICs consistently outperform CPU-based devices in terms of data-rate, cost and power; for example, a single switching chip today can process over 600Gb/s [2].

Scale: Because VINI and Emulab don't run new networking protocols on real hardware, they must always exist as a parallel testbed, which limits their scale. It would, for example, be prohibitively expensive to build a VINI or Emulab testbed to evaluate data-center-scale experiments requiring thousands or tens of thousands of switches, each with a capacity of hundreds of gigabits per second. VINI's geographic scope is limited by the locations willing to host special servers (42 today). Without enormous investment, it is unlikely to grow to global scale. Emulab can grow larger, as it is housed under one roof, but is still unlikely to grow to a size representative of a large network.

Technology transfer: An experiment running on a network of CPUs takes considerable effort to transfer to specialized hardware; the development styles are quite different, and the development cycle of hardware takes many years and requires many millions of dollars.

But perhaps the biggest limitation of a dedicated testbed is that it requires special infrastructure: equipment has to be developed, deployed, maintained and supported; and when the equipment is obsolete it needs to be updated. Networking testbeds rarely last more than one generation of technology, and so the immense engineering effort is quickly lost.

Our goal is to solve this problem. We set out to answer the following question: can we build a testbed that is embedded into every switch and router of the production network (in college campuses, data centers, WANs, enterprises, WiFi networks, and so on), so that the testbed would automatically scale with the global network, riding on its coat-tails *with no additional hardware*? If this were possible, then our college campus networks—for example—interconnected as they are by worldwide backbones, could be used simultaneously for production traffic and new WAN routing experiments; similarly, an existing data center with thousands of switches can be used to try out new routing schemes. Many of the goals of programs like GENI and FIRE could be met without needing dedicated network infrastructure.

In this paper, we introduce FlowVisor which aims to turn the production network itself into a testbed (Figure 1). That is, FlowVisor allows experimenters to evaluate ideas directly in the production network (not running in a dedicated testbed alongside it) by "slicing" the hardware already installed. Experimenters try out their ideas in an isolated slice, without the need for dedicated servers or specialized hardware.

1.1 Contributions.

We believe our work makes five main contributions:

Runs on deployed hardware and at real line-rates. FlowVisor introduces a software *slicing layer* between the forwarding and control planes on network devices. While FlowVisor could slice any control plane message format, in practice we implement the slicing layer with OpenFlow [16]. To our knowledge, no previously proposed slicing mechanism allows a user-defined control plane to control the forwarding in deployed production hardware. Note that this would not be possible with VLANs—while they crudely separate classes of traffic, they provide no means to control the forwarding plane. We describe the slicing layer in §2 and FlowVisor's architecture in §3.

Allows real users to opt-in on a per-flow basis. FlowVisor has a policy language that maps flows to slices. By modifying this mapping, users can easily try new services, and experimenters can entice users to bring real traffic. We describe the rules for mapping flows to slices in §3.2.

Ports easily to non-sliced networks. FlowVisor (and its slicing) is transparent to both data and control planes, and therefore, the control logic is unaware of the slicing layer. This property provides a direct path for vendor adoption. In our OpenFlow-based implementation, neither the OpenFlow switches or the controllers need be modified to interoperate with FlowVisor (§3.3).

Enforces strong *isolation* between slices. FlowVisor blocks and rewrites control messages as they cross the slicing layer. Actions of one slice are prevented from affecting another, allowing experiments to safely coexist with real production traffic. We describe the details of the isolation mechanisms in §4 and evaluate their effectiveness in §5.

Operates on deployed networks FlowVisor has been deployed in our production campus network for the last 7 months. Our deployment consists of 20+ users, 40+ network devices, a production traffic slice, and four standing experimental slices. In §6, we describe our current deployment and future plans to expand into seven other campus networks and two research backbones in the coming year.

2 Slicing Control & Data Planes

On today's commercial switches and routers, the control plane and data planes are usually logically distinct but physically co-located. The control plane creates and populates the data plane with forwarding rules, which the data plane enforces. In a nutshell, FlowVisor assumes that the control plane can be separated from the data plane, and it then *slices* the communication between them. This slicing approach can work several ways: for example, there might already be a clean interface between the control and data planes *inside* the switch. More likely, they are separated by a common protocol (*e.g.*, OpenFlow [16] or ForCes [7]). In either case, FlowVisor sits *between* the control and data planes, and from this vantage point enables a single data plane to be controlled by multiple control planes—each belonging to a separate experiment.

With FlowVisor, each experiment runs in their own slice of the network. A researcher, Bob, begins by requesting a network slice from Alice, his network administrator. The request specifies his requirements including topology, bandwidth, and the set of traffic—defined by a set of flows, or *flowspace*—that the slice controls. Within his slice, Bob has his own control plane where he puts the control logic that defines how packets are forwarded and rewritten in his experiment. For example, imagine that Bob wants to create a new *http* load-balancer to spread port 80 traffic over multiple web servers. He requests a slice: its topology should encompass the web servers, and its flowspace should include all flows with port 80. He is allocated a control plane where he adds his load-balancing logic to control how flows are routed in the data plane. He may advertise his new service so as to attract users. Interested users "opt-in" by contacting their network administrator to add a subset of their flows to the flowspace of Bob's slice.

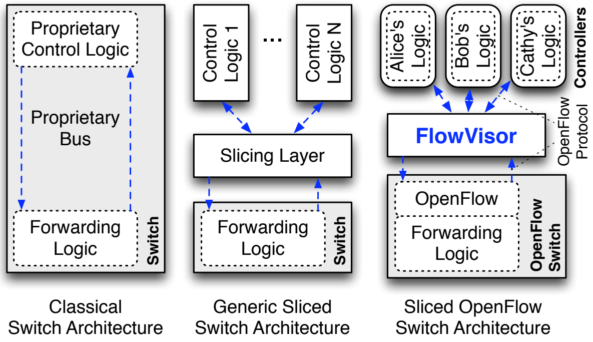

Figure 2: Classical network device architectures have distinct forwarding and control logic elements (left). By adding a transparent *slicing layer* between the forwarding and control elements, FlowVisor allows multiple control logics to manage the same forwarding element (middle). In implementation, FlowVisor uses OpenFlow and sits between an OpenFlow switch—the forwarding element—and multiple OpenFlow controllers—the control logic (right).

In this example, FlowVisor allocates a control plane for Bob, and allows him to control his flows (but no others) in the data plane. Any events associated with his flows (*e.g.* when a new flow starts) are sent to his control plane. FlowVisor enforces his slice's topology by only allowing him to control switches within his slice.

FlowVisor slices the network along multiple dimensions, including topology, bandwidth, and forwarding table entries. Slices are isolated from each other, so that actions in one slice—be they faulty, malicious, or otherwise—do not impact other slices.

2.1 Slicing OpenFlow

While architecturally FlowVisor can slice any data plane/control plane communication channel, we built our prototype on top of OpenFlow.

OpenFlow [16, 18] is an open standard that allows researchers to directly control the way packets are routed in the network. As described above, 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 protocol and 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 2, right). The OpenFlow protocol abstracts

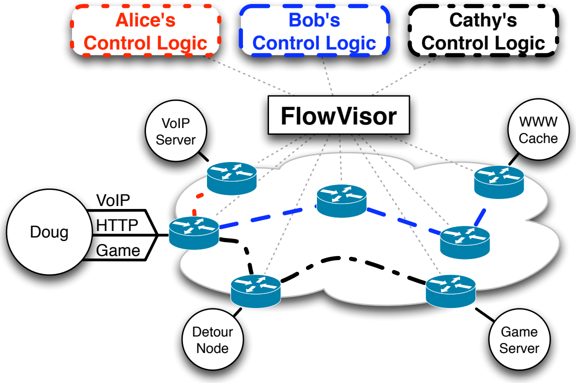

Figure 3: FlowVisor allows users (Doug) to delegate control of subsets of their traffic to distinct researchers (Alice, Bob, Cathy). Each research experiment runs in its own, isolated network slice.

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. If the packet doesn't match any entry, the packet is queued and a new flow event is sent across the network to the OpenFlow controller. The controller responds by adding a new rule to the flow table to handle the queued packet. Subsequent packets in the same flow will be handled without contacting the controller. Thus, the external controller need only be contacted for the first packet in a flow; subsequent packets are forwarded 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.

Note that while OpenFlow allows researchers to experiment with new network protocols on deployed hardware, only a single researcher can use/control an OpenFlow-enabled network at a time. As a result, without FlowVisor, OpenFlow-based research is limited to isolated testbeds, limiting its scope and realism. Thus, FlowVisor's ability to slice a production network is an orthogonal and indepenent contribution to OpenFlow-like software-defined networks.

3 FlowVisor Design

To restate our main goal, FlowVisor aims to use the production network as a testbed. In operation, the FlowVisor slices the network by slicing each of the network's corresponding packet forwarding devices (*e.g.*, switches and routers) and links (Figure 3).

With the FlowVisor,

- Network resources are sliced in terms of their bandwidth, topology, forward table entries, and device CPU (§3.1).

- Each slice has control over a set of flows, called its *flowspace*. Users can arbitrarily add (opt-in) and remove (opt-out) their own flows from a slice's flowspace at anytime (§3.2).

- Each slice has its own distinct, programmable control logic, that manages how packets are forwarded and rewritten for traffic in the slice's flowspace. In practice, each slice owner implements their slice-specific control logic as an OpenFlow controller. The FlowVisor interposes between data and control planes by proxying connections between OpenFlow switches and each slice controller (§3.3).

- Slices are defined using a slice definition policy language. The language specifies the slice's resource limits, flowspace, and controller's location in terms of IP and TCP port-pair (§3.4).

3.1 Slicing Network Resources

Slicing a network means correctly slicing all of the corresponding network resources. There are four primary slicing dimensions:

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

Bandwidth. Each slice has its own fraction of bandwidth on each link. Failure to isolate bandwidth would allow one slice to affect, or even starve, another slice's throughput.

Device CPU. Each slice is limited to what fraction of each device's CPU that it can consume. Switches and routers typically have very limited general purpose computational resources. Without proper CPU slicing, switches will stop forwarding slow-path packets (§5.3.2), drop statistics requests, and, most importantly, will stop processing updates to the forwarding table.

Forwarding Tables. Each slice has a finite quota of forwarding rules. Network devices typically support a finite

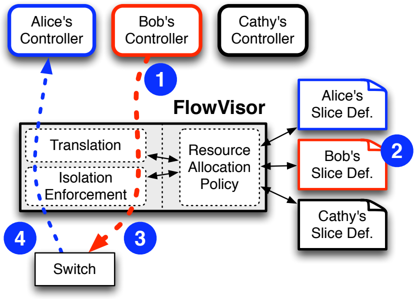

Figure 4: The FlowVisor intercepts OpenFlow messages from guest controllers (1) and, using the user's slicing policy (2), transparently rewrites (3) the message to control only a slice of the network. Messages from switches (4) are forwarded only to guests if it matches their slice policy.

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.

3.2 Flowspace and Opt-In

A slice controls a subset of traffic in the network. The subset is defined by a collection of packet headers that form a well-defined (but not necessarily contiguous) subspace of the entire space of possible packet headers. Abstractly, if packet headers have n bits, then the set of all possible packet header forms an n-dimensional space. An arriving packet is a single point in that space representing all packets with the same header. Similar to the geometric representation used to describe access control lists for packet classification [14], we use this abstraction to partition the space into regions (flowspace) and map those regions to slices.

The flowspace abstraction helps us manage users who opt-in. To opt-in to a new experiment or service, users signal to the network administrator that they would like to add a subset of their flows to a slice's flowspace. Users can precisely decide their level of involvement in an experiment. For example, one user might opt-in all of their traffic to a single experiment, while another user might just opt-in traffic for one application (e.g., port 80 for HTTP), or even just a specific flow (by exactly specifying all of the fields of a header). In our prototype the opt-in process is manual; but in a ideal system, the user would be authenticated and their request checked automatically against a policy.

For the purposes of testbed we concluded flow-level opt-in is adequate—in fact, it seems quite powerful. Another approach might be to opt-in individual packets, which would be more onerous.

3.3 Control Message Slicing

By design, FlowVisor is a slicing layer interposed between data and control planes of each device in the network. In implementation, FlowVisor acts as a transparent proxy between OpenFlow-enabled network devices (acting as dumb data planes) and multiple OpenFlow *slice* controllers (acting as programmable control logic—Figure 4). All OpenFlow messages between the switch and the controller are sent through FlowVisor. FlowVisor uses the OpenFlow protocol to communicate upwards to the slice controllers and and downwards to OpenFlow switches. Because FlowVisor is transparent, the slice controllers require no modification and believe they are communicating directly with the switches.

We illustrate the FlowVisor's operation by extending the example from §2 (Figure 4). Recall that a researcher, Bob, has created a slice that is an HTTP proxy designed to spread all HTTP traffic over a set of web servers. While the controller will work on any HTTP traffic, Bob's FlowVisor policy slices the network so that he only sees traffic from users that have opted-in to his slice. His slice controller doesn't know the network has been sliced, so doesn't realize it only sees a subset of the HTTP traffic. The slice controller thinks it can control, i.e., insert flow entries for, all HTTP traffic from any user. 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 4-1), examines Bob's slice policy (Figure 4-2), and rewrites the entry to include only traffic from the allowed source (Figure 4-3). 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 4-4) are only forwarded to guest controllers whose flowspace match the message. That is, it will only be forwarded to Bob if the new flow is HTTP traffic from a user that has opted-in to his slice.

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 slice controller to switch, FlowVisor ensures that the message acts only on traffic within the resources assigned to the slice. For a message in the opposite direction (switch to controller), the FlowVisor examines the message content to infer the corresponding slice(s) to which the message should be forwarded. Slice controllers only receive messages that are relevant to their

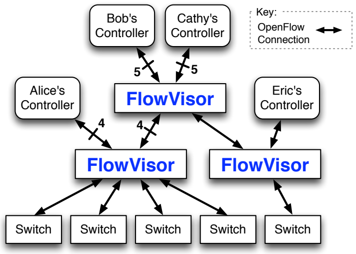

Figure 5: FlowVisor can trivially recursively slice an already sliced network, creating hierarchies of FlowVisors.

network slice. Thus, from a slice 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 slice multiple physical switches, and even re-slice an already sliced network (Figure 5) .

3.4 Slice Definition Policy

The slice policy defines the network resources, flowspace, and OpenFlow slice controller allocated to each slice. Each policy is described by a text configuration file—one file per slice. In terms of resources, the policy defines the fraction of total link bandwidth available to this slice (§4.3) and the 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. The rules define the flowspace a slice controls. Read-only rules allow slices to receive OpenFlow control messages and query switch statistics, but not to write entries into the forwarding table. Rules are allowed to overlap, as described in the example below.

Let's take a look at an example set of rules. Alice, the network administrator, wants to allow Bob to conduct an HTTP load-balancing experiment. Bob has convinced some of his colleagues to opt-in to his experiment. Alice wants to maintain control of all traffic that is not part of Bob's experiment. She wants to passively monitor all network performance, to keep an eye on Bob and the production network.

Here is a set of rules Alice could install in the FlowVisor:

Bob's Experimental Network includes all HTTP traffic to/from users who opted into his experiment. Thus, his network is described by one rule per user: `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 modified to meet 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 slices. It has one rule, `Read-only: all`.

This rule-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 will create new resource allocation policies.

4 FlowVisor Implementation

We implemented FlowVisor in approximately 8000 lines of C and the code is publicly available for download from `www.openflow.org`. The notable parts of the implementation are the transparency and isolation mechanisms. Critical to its design, FlowVisor acts as a *transparent* slicing layer and enforces *isolation* between slices. In this section, we describe how FlowVisor rewrites control messages—both down to the forwarding plane and up to the control plane—to ensure both transparency and strong isolation. Because isolation mechanisms vary by resource, we describe each resource in turn: bandwidth, switch CPU, and forwarding table entries. In our deployment, we found that the switch CPU was the most constrained resource, so we devote particular care to describing its slicing mechanisms.

4.1 Messages to Control Plane

FlowVisor carefully rewrites messages from the OpenFlow switch to the slice controller to ensure transparency. First, FlowVisor only sends control plane messages to a slice controller if the source switch is actually in the slice's topology. Second, FlowVisor rewrites OpenFlow feature negotiation messages so that the slice controller only sees the physical switch ports that appear in the slice. Third, OpenFlow port up/port down messages are similarly pruned and only forwarded to the affected slices. Using these message rewriting techniques, FlowVisor can easily simulate network events, such as link and node failures.

4.2 Messages to Forwarding Plane

In the opposite direction, FlowVisor also rewrites messages from the slice controller to the OpenFlow switch. The most important messages to the forwarding plane were insertions and deletions to the forwarding table. Recall (§2.1) that in OpenFlow, forwarding rules consist of a flow rule definition, i.e., a bit pattern, and a set of actions. To ensure both transparency and isolation, the FlowVisor rewrites both the flow definition and the set of actions so that they do not violate the slice's definition.

Given a forwarding rule modification, the FlowVisor rewrites the flow definition to intersect with the slice's flowspace. For example, Bob's flowspace gives him control over HTTP traffic for the set of users—e.g., users Doug and Eric—that have opted into his experiment. If Bob's slice controller tried to create a rule that affected all of Doug's traffic (HTTP and non-HTTP), then the FlowVisor would rewrite the rule to only affect the intersection, i.e., only Doug's HTTP traffic. If the intersection between the desired rule and the slice definition is null, e.g., Bob tried to affect traffic outside of his slice, e.g.., Doug's non-HTTP traffic, then the FlowVisor would drop the control message and return an error to Bob's controller. Because flowspaces are not necessarily contiguous, the intersection between the desired rule and the slice's flowspace may result in a single rule being expanded into multiple rules. For example, if Bob tried to affect all traffic in the system in a single rule, the FlowVisor would transparently expand the single rule in to two rules: one for each of Doug's and Eric's HTTP traffic.

FlowVisor also rewrites the lists of actions in a forwarding rule. For example, if Bob creates a rule to send out all ports, the rule is rewritten to send to just the subset of ports in Bob's slice. If Bob tries to send out a port that is not in his slice, the FlowVisor returns a "action is invalid" error (recall that from above, Bob's controller only discovers the ports that do exist in his slice, so only in error would he use a port outside his slice).

4.3 Bandwidth Isolation

Typically, even relatively modest commodity network hardware has some capability for basic bandwidth isolation [13]. The most recent versions of OpenFlow expose native bandwidth slicing capabilities in the form of per-port queues. The FlowVisor creates a per-slice queue on each port on the switch. The queue is configured for a fraction of link bandwidth, as defined in the slice definition. To enforce bandwidth isolation, the FlowVisor rewrites all slice forwarding table additions from "send out port X" to "send out queue Y on port X ", where Y is a slice-specific queue ID. Thus, all traffic from a given slice is mapped to the traffic class specified by the resource allocation policy. While any queuing discipline can be used (weighted fair queuing, deficit round robin, strict partition, etc.), in implementation, FlowVisor uses minimum bandwidth queues. That is, a slice configured for $X\%$ of bandwidth will receive at least $X\%$ and possibly more if the link is under-utilized. We choose minimum bandwidth queues to avoid issues of bandwidth fragmentation. We evaluate the effectiveness of bandwidth isolation in §5.

4.4 Device CPU Isolation

CPUs on commodity network hardware are typically low-power embedded processors and are easily overloaded. The problem is that in most hardware, a highly-loaded switch CPU will significantly disrupt the network. For example, when a CPU becomes overloaded, hardware forwarding will continue, but the switch will stop responding to OpenFlow requests, which causes the forwarding tables to enter an inconsistent state where routing loops become possible, and the network can quickly become unusable.

Many of the CPU-isolation mechanisms presented are not inherent to FlowVisor's design, but rather a work-around to deal with the existing hardware abstraction exposed by OpenFlow. 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 rate-limiters to enforce CPU isolation between OpenFlow and non-OpenFlow VLANs. Adding these features to OpenFlow is ongoing.

There are four main sources of load on a switch CPU: (1) generating new flow messages, (2) handling 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 rate limits the new flow message arrival rate. In implementation, the FlowVisor tracks the new flow message arrival rate for each slice, and if it exceeds some threshold, the FlowVisor inserts a forwarding rule to drop the offending packets for a short period.

For example, the FlowVisor keeps a token-bucket style counter for each flow space rule ("Bob's slice gets (1) all HTTP traffic and (2) all HTTPS traffic", i.e., two rules/counters). Each time the FlowVisor receives a new flow event, the token bucket that matches the flow gets decremented (for Bob's slice, packets that match HTTP count against token bucket #1, packets that match HTTPS count against #2). Once the bucket is emptied, the FlowVisor inserts a lowest-priority rule into the switch to drop all packets in that flowspace rule, i.e., from the example, if the token bucket corresponding to HTTPS is emptied, then the flowvisor will cause the switch to drop all HTTPS packets—without generating new flow events. The rule is set to expire in 1 second, so it is effectively a very coarse rate limiter. In practice, if a slice has control over "all traffic", this mechanism effectively blocks all new flow events from saturating the switch CPU or going to the controller, while allowing all existing flows to continue without change. We discuss the effectiveness of this technique in §5.3.2.

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.

This is because, in implementations, most switches only implement a subset of OpenFlow's functionality in their hardware. For example, the ASICs on most switches do not support sending one packet out exactly two ports (they support unicast and broadcast, but not in between). To emulate this behavior, the switches actually process these types of flows in their local CPUs, i.e,. on their slow path. Unfortunately, as mentioned above, these are embedded CPUs and are not as powerful as those on, for example, commodity PCs.

FlowVisor prevents slice 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 is 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 switch's internal function.

4.5 Flow Entry 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. When a guest controller exceeds its flow entry limit, any new rule insertions received a "table full" error message.

5 Evaluation

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

5.1 Scalability

A single FlowVisor instance scales well enough to serve our entire 40+ switch, 7 slice deployment with minimal load. As a result, we create an artificially high workload to evaluate our implementation's scaling limits. The FlowVisor's workload is characterized by the number of switches, slices, and flowspace rules per slice as well as the rate of new flow messages. We present the results for two types of workloads: one that matches what we **observe** from our deployment (1 slice, 35 rules per slice, 28 switches[1], 1.55 new flows per second per switch) and the other a **synthetic** workload (10 switches, 100 new flows per second per switch, 1 slice, 1000 rules per slice) designed to stress the system. In each graph, we fix three variables according to their workload and vary the forth.

Our evaluation measured FlowVisor's CPU utilization using a custom script. The script creates a configurable number of OpenFlow connections to the FlowVisor, and each connections simulates a switch that sends new flow messages to the FlowVisor at a prescribed rate. With each experiment, we configured the FlowVisor's number of slices and flowspace rules per slice. The new flow messages were carefully crafted to match only the last rule of each slice, causing the worst case behavior in the FlowVisor's linear search of the flowspace rules. Each test was run for 5 minutes and we recorded the CPU utilization of the FlowVisor process once per second, so each result is the average of 300 samples (shown

[1] This particular measurement did not include all of the switches in out network.

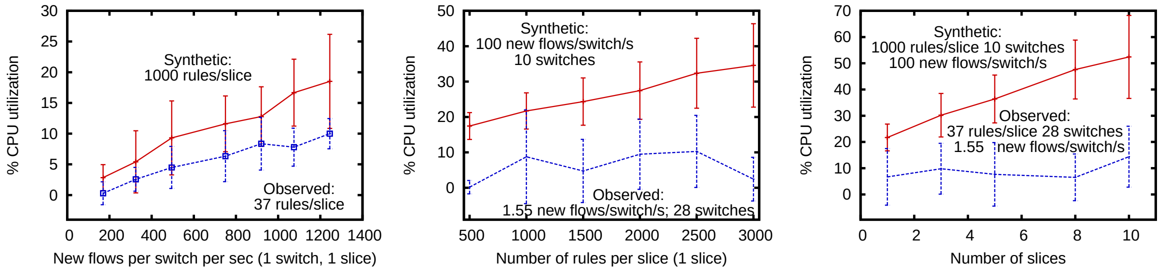

Figure 6: FlowVisor scales linearly with new flow rate, number of slices, switches, and flowspace rules. We generate high synthetic workloads to explore the scalability because the workloads observed in our deployment were non-taxing.

with one standard deviation). The FlowVisor ran on a quad-core Intel Xeon 3GHz system running 32-bit Debian Linux 5.0 (Lenny).

Our results with the synthetically high workload show that the FlowVisor's CPU load scales linearly in each of these four workload dimensions (as summarized in Figure 6). The result is promising, but not surprising. Intuitively, the FlowVisor can process a fixed number of OpenFlow messages per second (the product of number of switches by new flow rate) and each message must be matched against each rule of each slice, so the total load is approximately the product of the four workload variables. The synthetic workload with 1,000 new flows/s (10 switches by 100 new flows/s) is comparable to the peak rate of published real-world enterprise networks [20]: an 8,000 host network generated a peak rate of 1,200 new flows per second. Thus, we believe that a single FlowVisor instance could manage a large enterprise network. By contrast, our observed workload fluctuated between 0% and 10% CPU, roughly independent of the experimental variable. This validates our belief that our deployment can grow significantly using a single FlowVisor instance.

While our results show that FlowVisor scales well beyond our current requirements and workload, it is worth noting that it is possible to achieve even further scaling by moving to a multi-threaded implementation (the current implementation is single threaded) or even to multiple FlowVisor instances.

5.2 Performance Overhead

Adding an additional layer between control and data planes adds overhead to the system. However, as a result of our design, the FlowVisor does not add overhead to the data plane. 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 un-sliced rate. FlowVisor only adds overhead to actions that cross between the control and data plane layers.

To quantify this cross-layer overhead, we measure the increased response time for slice 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: the new flow and the port status request messages.

In OpenFlow, a switch sends a new flow message to its controller when an arriving packet does not match any existing forwarding rules. We examine the increased delay of the new flow message to better understand how the FlowVisor affects connection setup latency. In our experiment, we connect a machine with two interfaces to a switch. One interface sends a packet every 20ms (50 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 7(a)) show that the FlowVisor increases time from the switch to controller by an average of 16ms. 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 FlowVisor, so we believe that slice controllers in those environments will likely proactively insert flow entries into switches, avoiding this latency all together. We point out that the algorithm FlowVisor uses to process new flow messages is naive, and its run-time grows linearly with the number of flowspace rules (§5.1). We are yet to experiment with the many classification algorithms that can be expected to improve the lookup speed.

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 port 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, but expensive for the FlowVisor to process: it has to edit the

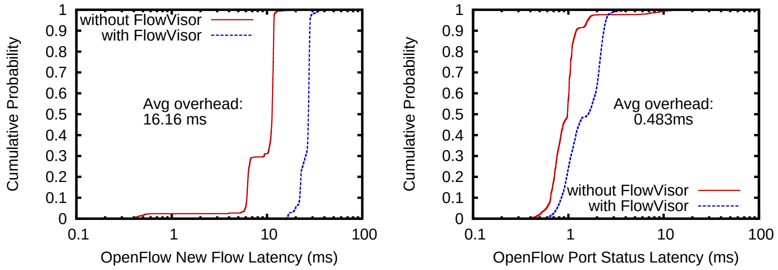

Figure 7: CDF of slicing overhead for OpenFlow new flow messages and port status requests.

message per slice to remove statistics for ports that do not appear in a sliced topology.

We wrote a special-purpose controller that sent approximately 200 port status 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 7(b)). 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 request handling.

5.3 Isolation

5.3.1 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 TCP-unfriendly 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 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 8(a)). With the bandwidth isolation disabled ("without Slicing"), 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 ("with 30/70% reservation"), 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 queue is designed.

5.3.2 Switch CPU

To quantify our ability to isolate the switch CPU resource, we show two experiments that monitor CPU-usage over time of a switch with and without isolation enabled. In the first experiment (Figure 8(b)), the OpenFlow controller maliciously sends port stats request messages (as above) at increasing speeds $(2, 4, 8 \ldots 1024$ requests per second). In our second experiment (Figure 8(c)), 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 switch's CPU utilization averaged over one second, 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(c) is a direct result of using null forwarding rules (§4.4) 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.

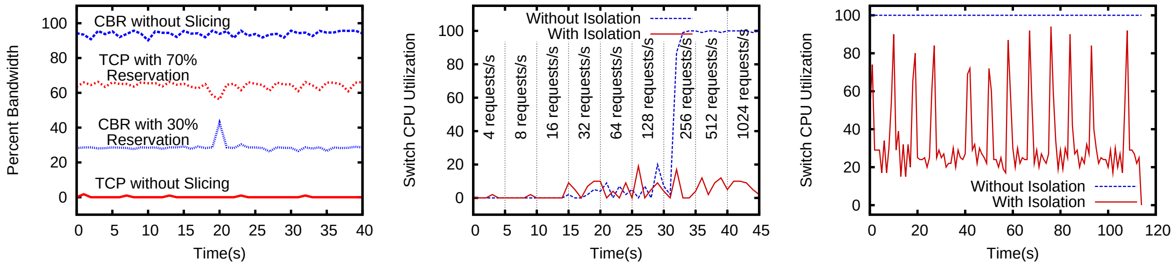

Figure 8: FlowVisor's bandwidth isolation prevents CBR traffic from starving TCP, and message throttling and new flow message rate limiting prevents CPU starvation.

6 Deployment Experience

To provide evidence that sliced experimental traffic can indeed co-exist with production traffic, we deployed FlowVisor on our production network. By "production", we refer to the network that the authors rely on to read their daily email, surf the web, etc. Additionally, six other campuses are currently using the FlowVisor as part of the GENI "meso-scale" infrastructure. In this section, we describe our experiences in deploying FlowVisor in our production network, its deployment in other campuses, and briefly describe the experiments that have run on the FlowVisor.

6.1 Stanford Deployment

At Stanford University, we have been running FlowVisor continuously on our production network since June 4th, 2009. Our network consists of 25+ users, 5 NEC IP8800 switches, 2 HP ProCurve 5400s, 30 wireless access points, 5 NetFPGA [15] cards acting as OpenFlow switches, and a WiMAX base station. Our physical network is effectively doubly sliced: first by VLANs and then by the FlowVisor. Our network trunks over 10 VLANs , including traffic for other research groups, but only three of those VLANs are OpenFlow-enabled. Of the three OpenFlow VLANs, two are sliced by FlowVisor. We maintain multiple OpenFlow VLANs and FlowVisor instances to allow FlowVisor development without impacting production traffic.

For each FlowVisor-sliced VLAN, all network devices point to a single FlowVisor instance, running on a 3.0GHz quad-core Intel Xeon with 2 GB of DRAM. 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 module to perform basic forwarding in the network. We will publish our slicing administration tools and debugging techniques.

6.2 Deploying on Other Networks

As part of the GENI "meso-scale" project, we also deployed FlowVisor onto test networks on six university campuses, including University of Washington, Wisconsin University, Princeton University, Indiana University, Clemson University, Rutgers University. In each network, we have a staged deployment plan with the eventual goal of extending the existing OpenFlow and FlowVisor test network to their production networks. Each network runs its own FlowVisor. Recently, at the 8th GENI Engingering Conference (GEC), we demonstrated how slices at each campus's network could be combined with tunnels to create a single wide-area network slice. Currently, we are in the process of extending the FlowVisor deployment into two backbone networks (Internet2 and National Lambda Rail), with the eventual goal of creating a large-scale end-to-end sliceable wide area network.

6.3 Slicing Experience

In our experience, the two largest causes of network instability were unexpected interactions with other deployed network devices and device CPU exhaustion. One problem we had was interacting with a virtual IP feature of the router in our building. This feature allows multiple physical interfaces to act as a single, logical interface for redundancy. In implementation, the router would reply to ARP requests with the MAC address of the logical interface but source packets from any of three different MAC addresses corresponding to the physical interfaces. As a result, we had to revise the flowspace assigned to the production slice to include all four MAC addresses. Another aspect that we did not anticipate is the amount of broadcast traffic emitted from non-OpenFlow devices. It is quite common for a device to periodically send broadcast LLDP, Spanning Tree, and other packets. The level of broadcast traffic on the network made debugging more difficult and could cause loops if our OpenFlow-based loop detection/spanning tree algorithms did not match the non-OpenFlow-bases spanning tree.

Another issue was the interaction between OpenFlow and device CPU usage. As discussed earlier (§ 4.4), the most frequent form of slice isolation violations occurred with device CPU. The main form of isolation violation occurred when one slice would insert a forwarding rule that could only be handled via the switch's slow path and, as a result, would push the CPU utilization to 100%, preventing slices from updating their forwarding rules. We also found that the cost to process an OpenFlow message varied significantly by type and by OpenFlow implementation particularly with stats requests, e.g., the OpenFlow aggregate stats command consumed more CPU than an OpenFlow port stats command, but not on all implementations. As part of our future work, we plan to compute a per-message type costs to each OpenFlow request to more precisely slice device CPU. Additionally, the upcoming OpenFlow version 1.1 will add support for rate limiting messages coming from the fast to slow paths.

6.4 Experiments

We've demonstrated that FlowVisor supports a wide variety of network experiments. On our production network, we ran four networking experiments, each in its own slice. All four experiments, including a network load-balancer [12], wireless streaming video [26], traffic engineering, and a hardware prototyping experiment [9], were built on top of NOX [11]. As part of the 7th GENI Engingeering Conference, each of the seven campuses demonstrated their own, locally designed experiments, running in a FlowVisor-enabled slice of the network. Our hope is that the FlowVisor will continue to allow researchers to run novel experiments in their own networks.

7 Related Work

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

The community has benefited from a number of testbeds for performing large-scale experiments. The two most widely used are PlanetLab [21] and Emulab [25]. 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. ShadowNet [3] exposes virtualization features of specific high-end routers, but does not provide per-flow forwarding control or user opt-in. VINI [1], 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.* [5]).

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 we slice the hardware forwarding paths of unmodified commercial network gear.

Supercharged PlanetLab [23] is a network experimentation platform designed around CPUs and NPUs (network processors). NPUs can provide high performance and isolation while allowing for sophisticated per-packet processing. In contrast, our work forgoes the ability to perform arbitrary per-packet computation in order to work on unmodified hardware.

VLANs [4] are widely used for segmentation and isolation in networks. VLANs slice Ethernet L2 broadcast domains by decoupling virtual links from 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. VLANs use a specific control logic (L2 forwarding and learning over a spanning tree). FlowVisor, on the other hand, allows users to define their own control logic. It also supports a more flexible method for defining the traffic that is in a slice, and the way users opt in. For example, with FlowVisor a user could opt-in to two different slices, whereas with VLANs their traffic would all be allocated to a single slice at Layer 2.

Perhaps the most similar to FlowVisor is the Prospero ATM Switch Divider Controller [24]. Prospero uses a hardware abstraction interface, Ariel, to allow multiple control planes to operate on the same data plane. While architecturally similar to our design, Prospero slices individual ATM switches where FlowVisor has a centralized view and can thus create a slice of the entire network. Further, Ariel provides the ability to match on ATM-related fields (e.g., VCI/VPI) where OpenFlow can match on any combination of 12-fields spanning layers one through four. This additional capability is critical for our notion of flow-level opt-in.

8 Trade-offs and Caveats

The FlowVisor *approach* is extremely general—it simply states that if we can insert a slicing layer between the control and data planes of switches and routers, then we can perform experiments in the production network. In principle, the experimenter can exploit any capability of the data plane, so long as it is made available to them.

Our prototype of FlowVisor is based on OpenFlow, which makes very *few* of the hardware capabilities available—which limits the flexibility. Most switch and router hardware can do a lot more than is exposed via OpenFlow (*e.g.* dozens of different packet scheduling policies, encapsulation into VLANs, VPNs, GRE tunnels, MPLS, and so on). OpenFlow makes a trade-off: it only exposes a lowest common denominator that is present in all switches in return for a common vendor-agnostic interface. So far, this minimal set has met the needs of early experimenters—there appears to be a basic set of primitive "plumbing" actions that are sufficient for a wide array of experiments, and over time we would expect the OpenFlow specification to evolve to be "just enough", like the RISC instruction set in CPUs. In addition to the diverse set of experiments we have created, others have created experiments for data center network schemes (such as VL2 and Portland), new routing schemes, home network managers, mobility managers, and so on.

However, there will always be experimenters who need more control over individual packets. They might want to use features of the hardware not exposed by OpenFlow; or they might want full programmatic control, not available in any commercial hardware. The first case is a little easier to handle, because a switch or router manufacturer can expose more features to the experimenter if they choose, either by vendor-specific extensions to OpenFlow and FlowVisor, or by allowing flows to be sent to a logical internal port that, in turn, processes the packets in a pre-defined box-specific way.[2]

But if an experiment needs a way to modify packets *arbitrarily*, the researcher needs a different box. If the experiment calls for arbitrary processing in novel ways at *every* switch in the network, then OpenFlow is probably not the right interface, and our prototype is unlikely to be of much use. If the experiment only needs processing at some parts of the network (*e.g.* to do deep packet inspection, or payload processing) then the researcher can route their flows through some number of special middle-boxes or way-points. The middle-boxes could be conventional servers, NPUs [23], programmable hardware [15], or custom hardware. The good thing is that these boxes can be placed anywhere, and the flows belonging to a slice can be routed through them - including all the flows from users who opt in. In the end, the value of FlowVisor to the researcher will depend on how many middle-boxes the experiment needs to be realistic—just a few and it may be worth it; if it needs hundreds or thousands then FlowVisor is providing very little value.

A second limitation of our prototype is the ability to create arbitrary topologies. If a physical switch is to appear multiple times in a slice's topology (*i.e.* to create a virtual topology larger than the physical topology), there is currently no standardized way to do this. The hardware needs to allow packets to loop back, and pass through the switch multiple times. In fact, most—but not all—switch hardware allows this. At some later date we expect this will be exposed via OpenFlow, but in the meantime it remains a limitation.

9 Conclusion

Put bluntly, the problem with testbeds is that they are *testbeds*. If we could test new ideas at scale, with real users, traffic and topologies, without building a testbed, then life would be much simpler. Clearly this isn't the case today: testbeds need to be built, maintained, and are expensive to deploy at scale. They become obsolete quickly, and many university machine rooms have outdated testbed equipment lying around unused.

By definition, a testbed is not the real network: therefore, we try to embed testbeds into the network by slicing the hardware. This paper described our first attempt towards embedding a testbed in the network. While not yet bullet-proof, we believe that our approach of slicing the communication between the control and data planes shows promise. Our current implementation is limited to controlling the abstraction of the forwarding element exposed by OpenFlow. We believe that exposing more fine-grained control of the forwarding elements will allow us to solve the remaining isolation issues (e.g., device cpu)—ideally with the help of the broader community. If we *can* perfect isolation, then several good things happen: researchers could validate their ideas at scale and with greater realism, the industry could perform safer quality assurance of new products, and finally, network operators could run multiple versions of the networks in parallel, allowing them to roll back to known good states.

Acknowledgments

We would like to thank Jennifer Rexford, Srinivasan Seetharaman, our shepherd Randy Katz, and the anonymous reviewers for their helpful comments and insight.

References

[1] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In vini veritas: realistic and controlled network experimentation. In *SIGCOMM '06*, pages 3–14, New York, NY, USA, 2006. ACM.

[2] For example, this is how some OpenFlow switches implement VPN tunnels today.

[2] BCM88130 - 630-Gbps High-Performance Packet Switch Fabric. http://www.broadcom.com/products/Switching/Carrier-and-Service-Provider/BCM88130.

[3] X. Chen, Z. M. Mao, and J. V. der Merwe. Shadownet: A platform for rapid and safe network evolution. In *Proceedings of USENIX Anual Technical Conference (USENIX'09)*. USENIX, 2009.

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

[5] 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.

[6] FIRE - Future Internet Research & Experimentation. http://cordis.europa.eu/fp7/ict/fire/.

[7] IETF ForCes. http://www.ietf.org/dyn/wg/charter/forces-charter.html.

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

[9] 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, 2009.

[10] The gigabit testbed initiative. http://www.cnri.reston.va.us/gigafr/.

[11] 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.

[12] 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.

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

[14] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM '98*, pages 203–214, New York, NY, USA, 1998. ACM.

[15] 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.

[16] 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.

[17] Ns2 network simulator. http://www.isi.edu/nsnam/ns/.

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

[19] Opnet technologies. http://www.opnet.com/.

[20] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *ACM Internet Measurement Conference*, 2005.

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

[22] J. Touch and S. Hotz. The x-bone. In *Proc. Global Internet Mini-Conference / Globecom*, 1998.

[23] J. S. Turner and et al. Supercharging planetlab: a high performance, multi-application, overlay network platform. In *SIGCOMM '07*, pages 85–96, New York, NY, USA, 2007. ACM.

[24] J. E. van der Merwe and I. M. Leslie. Switchlets and dynamic virtual atm networks. In *Proceedings of the fifth IFIP/IEEE international symposium on Integrated network management V : integrated management in a virtual world*, pages 355–368, London, UK, UK, 1997. Chapman & Hall, Ltd.

[25] B. White and J. L. et al. 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.

[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.