Virtualizing the Network Forwarding Plane

Martín Casado
Nicira

Teemu Koponen
Nicira

Rajiv Ramanathan
Google

Scott Shenker
UC Berkeley

1 Introduction

Modern system design often employs virtualization to decouple the system service model from its physical realization. Two common examples are the virtualization of computing resources through the use of virtual machines and the virtualization of disks by presenting logical volumes as the storage interface. The insertion of these abstraction layers allows operators great flexibility to achieve operational goals divorced from the underlying physical infrastructure. Today, workloads can be instantiated dynamically, expanded at runtime, migrated between physical servers (or geographic locations), and suspended if needed. Both computation and data can be replicated in real time across multiple physical hosts for purposes of high-availability within a single site, or disaster recovery across multiple sites.

Unfortunately, while computing and storage have fruitfully leveraged the virtualization paradigm, networking remains largely stuck in the physical world. As is clearly articulated in [8], networking has become a significant operational bottleneck. While the basic task of routing can be implemented on arbitrary topologies, the implementation of almost all other network services (*e.g.*, policy routes, ACLs, QoS, isolation domains) relies on topology-dependent configuration state [4, 9, 16]. Management of this configuration state is cumbersome and error prone – adding or replacing equipment, changing the topology, moving physical locations, or handling hardware failures often requires significant manual reconfiguration.

Virtualization is not foreign to networks, as networking has long supported virtualized primitives such as virtual links (tunnels), broadcast domains (VLANs), forwarding contexts (VRF), and component failover (*e.g.*, VRRP). However, these primitives have not significantly changed the operational model of networking, and operators continue to screen scrape CLIs with scripts in order to achieve a limited degree of automation. Thus, while computing and storage have both been greatly enhanced by the virtualization paradigm, networking has yet to break free from the physical infrastructure.

In this paper, we revisit the notion of network virtualization and describe how one can fully virtualize the network forwarding plane. Roughly, the idea is to introduce a new network-wide software layer that exposes one or more logical forwarding elements (similar to [10]). To software written to control the forwarding hardware (*i.e.*, control software such as routing algorithms) this new software layer looks like standard forwarding hardware. Instead of interfacing directly to the networking hardware as is done today, the control software reads and writes to these logical forwarding elements. This approach allows network state (forwarding and configuration) to be largely decoupled from the underlying hardware, paving the way for migration, failure resilience, and more complex state management (such as checkpointing and rollback).

While this approach is generally useful, by far the most compelling use cases demand a many-to-many mapping between the logical and physical forwarding elements: multiple logical forwarding elements may share the same physical router/switch, and a single logical forwarding element may span multiple physical routers/switches. In addition to providing basic support for function mobility (such as that provided by [16]), this enables a "scale-out" approach to network design, in which additional hardware scales-out a single control interface (for example an entire datacenter can be managed as a single switch instance). It also provides support for hardware sharing in multi-tenant environments, which is a crucial enabler for efficient networking in clouds. Yet despite the many benefits, there has been little treatment in the literature for a fully generalized virtualization of the network forwarding plane which would support such uses.

In this paper, we present a preliminary design and prototype implementation of such a system, describing how to leverage existing virtualization primitives in today's merchant silicon to achieve full data plane virtualization. Further, we describe a number of target applications for which our implementation is being designed.

In what follows, we describe in more detail what we mean by forwarding plane virtualization and how it differs from traditional approaches to network virtualization (§2). We then describe the design (§3) and implementation (§4) of our prototype. In §5 we describe uses cases, follow by related work in §6. Finally, we conclude with a discussion in §7.

2 Virtualization Revisited

As mentioned in the previous section, the disparate collection of virtualization primitives available in networks today (*e.g.*, VLANs, tunnels, VRF contexts) does not provide an adequate abstraction upon which to build topology-independent control software. Managing tunnels, VLAN configuration, and VRF contexts requires significant manual network management, and component failures can interfere with these virtual configurations by, for example, disrupting a tunnel end-point, eliminating VLAN configuration, or VRF forwarding state.

In our proposal, we take advantage of the sophisticated hardware support for these individual virtualization primitives to construct a more comprehensive network virtualization solution in which hardware can be treated generically as a resource pool of forwarding capacity, and hardware changes do not disrupt the logical view of the system.

The question, then, is what representation do we choose for the service model of the underlying physical infrastructure. We could have chosen various higher-level abstractions (such as the security configurations used in [9] or the access-control policies in [4]), but these are typically limited to particular service offerings. In order to provide application-independent virtualization, we chose to virtualize the forwarding plane. The forwarding plane is traditionally defined in hardware, so it is a slowly evolving interface whose basic abstractions (tables, ports, counters, and primitives for modifying and forwarding packets) have not changed significantly over time. In addition, this service interface is compatible with existing control software.

We have chosen the name "network hypervisor" for our proposed software layer to intentionally call to mind the concept of virtualization. Our proposed software layer serves much the same function as a hypervisor on a host: providing a logical service model to the software above, and then implementing the desired functionality on the hardware below. As we note in §6 and §7, the term "network virtualization" is often used to refer to carving a single physical network into several logical "slices". However, what we propose here is a strict superset of this slicing functionality, and provides a fuller virtualization of the network by providing a completely logical interface; network "slices" are not independent of the underlying physical infrastructure, but instead are a way of multiplexing that infrastructure.

3 Design

3.1 Overview

Virtualizing the network forwarding plane presents two major design challenges; the choice of the forwarding plane abstraction, and the technology needed to map the logical forwarding planes into the underlying physical hardware. We discuss these two issues in turn.

While each forwarding chipset differs in particulars, networking hardware typically exposes tables, counters, ports and port groups, and some control over packet manipulation and forwarding. Our hypervisor maintains these abstractions, providing the ability to create one or more logical (possibly interconnected) forwarding elements, where each forwarding element has a set of logical ports, a set of lookup tables, and some basic forwarding actions such as counters, forwarding, header-rewriting, and en/decapsulation. These elements also have associated capacities (*e.g.*, line speeds, cross-section bandwidth, table sizes). As with traditional physical networks, the control plane (*i.e.*, the software system used by operators to control the network) uses this logical abstraction to express the desired network functionality.

Note that this logical abstraction is not just used for static configuration of the network functionality, but instead can be used by network control software to implement sophisticated dynamic control over the network. For example, as we discuss later, one can use such an interface to turn a collection of physical routers into a single logical router that is participating in a routing protocol such as BGP. The "configuration" of the logical router is therefore programmatically determined by the implementation of BGP sitting on top of this logical interface.

To map the logical network state into the underlying hardware interface, we rely on a recent line of research [6, 7, 11, 12] that provides global network control. While there are many approaches to building such a system, our prototype is implemented using a control plane that provides a full view of the physical network topology. Our network hypervisor is thereby given two network views: from above it is given (by the control plane) a logical network view of the desired functionality, and from below it is given (by a centralized network management system) a view of the physical network topology. The job of the network hypervisor is to determine how to implement the desired logical functionality through configuration of the physical network. Thus, the network hypervisor should be seen as the point where the *logical* network is mapped into the *physical* network.

3.2 Components

In our proposal the network can be thought of as having several distinct logical layers:

- *Control Plane:* This refers to the basic mechanisms used to express the desired network functionality, through either manual configuration or programmatic control (as in routing algorithms).
- *Logical Forwarding Plane:* This is the logical abstraction of the network.
- *Network Hypervisor:* The network hypervisor, which is the subject of this paper, takes the logical forwarding plane and maps it into the underlying physical hardware.
- *Physical Forwarding Plane:* This refers to the set of physical network forwarding elements.

There is one other general concept that is useful in describing our system: that of a *logical context*. As a packet traverses the network, it can be thought of as moving in both the logical forwarding plane and the physical forwarding plane. When a switch is making a decision (based on its physical tables and the packet header) about how to forward a packet, it is often useful to know where that packet is in the logical forwarding plane: we call this information the *logical context* of the packet.

We now describe the logical forwarding plane, the physical forwarding plane, and the network hypervisor in more detail. We do not describe the control plane mechanisms, which determine the specification of the logical forwarding plane, in our description below.

3.3 Logical Forwarding Plane

For simplicity, in what follows we focus on the case where the logical forwarding plane consists of a single logical forwarding element. The interface of this logical element includes:

- *Lookup tables:* The logical forwarding element will expose one or more forwarding tables. Typically this will include L2, L3, and ACL tables. Our implementation is designed around OpenFlow [12] so we adopt a more generalized table structure built around a pipeline of TCAMs with forwarding actions specified for each rule. The forwarding actions correspond to the actions available in the physical forwarding plane, notably header overwriting, enqueuing, filtering, and multicast groupings. This structure provides support for forwarding rules, ACLs, SPAN, and other primitives.[1]

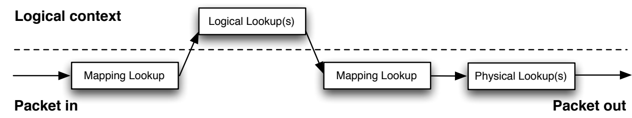
- *Ports:* The logical forwarding element contains a set of logical ports. These ports can be bound to physical ports, or to other port abstractions such as virtual machine interfaces, VLANs, or tunnels. Ports may appear and leave dynamically as they are either administratively added, or the component they are bound to fails or leaves. Logical ports maintain much of the same qualities of their physical analogs including rx/tx counters, MTU, speed, error counters, and carrier signal.

We chose this interface to be both familiar and expressive, but it also has the added advantage of being compatible with current hardware capabilities, so that it can be efficiently implemented. Moreover, it is compatible with current control plane mechanisms, making integration easier.

3.4 Physical Forwarding Plane

We assume that the forwarding elements are traditional hardware switches with standard forwarding silicon. As we discuss below, the network hypervisor is responsible

[1]Under this model, it is possible to support recursive virtual instances.



1: Packet forwarding pipeline in a physical switch.

for configuring the physical forwarding elements so that the network implements the desired behavior as specified by the logical forwarding plane. In order for the physical forwarding elements to carry out their assigned tasks, they must do the following for each packet: (*i*) map the incoming packet to the correct logical context, (*ii*) make a *logical* forwarding decision, (*iii*) map the logical forwarding decision back to the physical next-hop address, and (*iv*) make a *physical* forwarding decision in order to send packets to the physical next hop. Figure 1 shows these steps which we describe in more detail below.

Mapping packets to logical context. Multiple logical forwarding elements may be multiplexed over the same physical switch. Thus, on ingress, a packet must be mapped to the correct logical context. It may be the case that the current switch does not contain the logical forwarding state for a given packet, in which case the switch simply performs a physical forwarding decision. Also, if all the physical switches are implementing only a single logical forwarding element, the mapping becomes a no-op because logical addressing may be used in the physical network.

Conceptually, it does not matter which field(s) are used to map a packet to a logical context. It could be, for example, an identifying tag such as an MPLS header, or the ingress port. However, in order to provide transparency to end systems, the tag used for identifying logical contexts must not be exposed to the systems connecting to the logical switch. In general, this means that the first physical switch receiving a packet tags it to mark the context, and the last switch removes the tag. How the first tag is chosen depends largely on the deployment environment, we discuss this further in Section 4.

Logical forwarding. Once a packet is mapped to its logical context, the physical switch performs a forwarding decision which is only meaningful within the logical context. This could be, for example, an L2 lookup for the logical switch or a sequence of lookups required for a logical L3 router. However, if the physical switch executing the logical decision does not have enough capacity to have all the logical state, the logical decision executed at that switch may be only a step in the overall logical decision that needs be executed; in such a case, the packet may require further logical processing before leaving the logical forwarding element.

Mapping logical decision to physical. The result of a logical forwarding decision (assuming the packet wasn't dropped) is one or more egress ports on the logical forwarding element. Once these are determined, the network must send the packets to the physical objects to which these egress ports

are bound. This could be, for example, a physical port on another physical switch, or a virtual port of a virtual machine on a different physical server.

Thus, the network must map the logical egress port to the physical next hop. In our design, the logical and physical networks share distinct (though potentially overlapping) address spaces. Thus, once the physical address is found for the next hop, the (logical) packet must be encapsulated to be transferred to the next hop physical address.

If a logical forwarding decision is distributed across multiple physical components, the "next hop" will be the next physical component that will continue to execute the logical forwarding decision rather than a logical egress port.

Physical forwarding. Finally, the physical forwarding decision is responsible for forwarding the packet out of the correct physical egress port based on the physical address determined by the previous mapping step. This requires a third (or more) lookup over the new physical header (which we assume was created in the previous step).

It is worthwhile to note that if the physical switches of the network have only one logical context, the previous two steps may become no-ops.

3.5 Network Hypervisor

Given the previous description of the physical network responsibilities, the following state must be maintained at each switch in the network: *(i)* a table to map incoming packets to their logical forwarding context, *(ii)* rules for logical forwarding decisions, *(iii)* a table to map a logical egress port to a physical location, and *(iv)* a physical forwarding table.

Standard routing protocols such as OSPF or ISIS are well suited for populating the physical forwarding tables. However, our system also requires some method for determining logical to physical mappings, and distributing the logical forwarding rules across the physical network. This is the responsibility of the network hypervisor.

The network hypervisor maintains a global view of all physical resources in the network and all configured logical forwarding planes. Its primary function is to map the logical forwarding plane to the underlying hardware efficiently. It must also maintain these mappings whenever the physical network changes (component addition, removal, or failure) or the logical forwarding plane is modified.

4 Prototype Implementation

There are a number of approaches which would be suitable for realizing the conceptual design presented in the previous section. In what follows, we describe our hypervisor implementation which we have built as a distributed system that manages the switch state through an OpenFlow-based protocol. While we have tested a prototype, the implementation of our final design is not yet complete. In what follows, we discuss some of the practical issues we have considered in our implementation.

4.1 Physical Switch

In our implementation we use L2 over GRE to provide the physical to logical and logical to physical mappings. The L2 packets are relevant within the logical context, and the GRE tunnels provide the physical transport. The lookup which maps the L2 packets to the tunnels is the logical forwarding decision and is populated by the network hypervisor. The decision about which physical port to send a tunneled packet out of is the physical forwarding decision, and is determined via a standard routing protocol. To reduce the number of active tunnels the system has to maintain, we further use tagging (VLAN or MPLS) within tunnels to indicate the logical context to which a packet belongs.

We note that all of these functions are available on merchant silicon chipsets today. Both Broadcom and Marvell have chipsets that support rule based tunnel lookup for L2 over L3.

4.2 Network Hypervisor

In our implementation, the network hypervisor is being built as a distributed system that operates as an OpenFlow controller.[2] The network hypervisor connects with every switch in the network and uses a simple discovery mechanism to create an in-memory graph of the network.

Given the network graph, it is the responsibility of the hypervisor to map the logical forwarding plane to the physical network, and to maintain these mappings as the physical network changes either through a hardware failure or component addition or removal. The hypervisor also provides an API which allows for the creation and configuration of logical forwarding elements as well as dictating where they tie into the physical network.

The job of the hypervisor is complicated by the fact that it must multiplex the logical context over limited physical bandwidth and lookup table space. We discuss each in turn.

- *Bisectional bandwidth:* Ideally, logical forwarding elements can be declared with a given bisectional bandwidth which is enforced by the hypervisor. This is difficult to maintain in general over arbitrary topologies. Our current implementation does not deal with this problem directly but rather relies on load balancing of flows at the physical layer to uniformly consume physical forwarding bandwidth. While it tries to be efficient and fair, our implementation does not guarantee a minimum bisectional bandwidth.

- *Port bandwidth:* A logical port may be implemented as a tunnel which traverses multiple physical elements. In order to provide minimum port speed guarantees, each element must be able to support the capacity and isolate it from other traffic. Limitations on the number of queues in standard switching silicon makes this difficult to enforce in general. Again, our naive

[2]We omit the discussion of required OpenFlow changes for brevity.

implementation assumes over-provisioned end-to-end physical bandwidth.

- *TCAM space:* When placing logical forwarding rules, the hypervisor must consider the finite capacity of the physical forwarding tables. Unfortunately, unlike virtual memory in operating systems, the extreme performance demands of some network environments limits the practicality of an on-demand approach to "paging-in" rules on packet misses (such as the approach used in [5]). Rather, the rules in a logical context should be in hardware somewhere within the network.

We further simplify the resource optimization problem by only placing logical forwarding at the edge of the network, leaving the interior of the network to operate as a simple physical backplane.[3] Thus, the logical forwarding capacity is limited to that available in the first hop switches.

5 Use Cases

We now describe some of the practical use cases for a network hypervisor.

Distributed virtual switch. End-host virtualization requires switching capability on a host (implemented in the host hypervisor [13]). This is generally realized as an L2 software switch (generally termed vswitch) which connects all co-resident VMs on a physical host. Even with the ability to control the network at the end-host, the dynamic nature of virtual environments (which may include end-host migration) makes network monitoring and configuration difficult. A preferable approach is to connect VMs to a *distributed* logical switch which is topology independent (as is done with VMware DVS, for example). Our approach trivially allows us to support a distributed virtual switch (provided the vswitches support the required table lookups). Further, our approach allows this to be extended to physical switches allowing for non-virtualized end-hosts to participate on logical topologies.

Scale-out carrier-grade router. Data-centers and the edges of the wire-area networks, require a level of capacity and fan-out in IP routers that is difficult to fulfill with commodity switching hardware today. Thus, expensive carrier-grade routers are still the only viable solution.

A network hypervisor based solution can replace a single carrier-grade router with a rack of commodity switches. In essence, the commodity switches are used to a form a high capacity switching backplane that the network hypervisor then represents as a single logical switch for a standard (open-source) IGP/BGP routing stack control plane implementation. In this model, logical ports correspond to the physical ports used to interconnect this single "router in a rack" to its next hops, and the routing stack pushes its FIB (we are not assuming per ingress port FIBs here for simplicity) to the forwarding table of the logical switch, which is then distributed to the physical switches by the network hypervisor. Port status information is mapped back to the logical ports and the routing stack communicates with its peer by sending/receiving over the logical ports.

The switches are interconnected with a fat-tree-like topology to achieve sufficient bisectional bandwidth, and therefore, the network hypervisor has a simple task to allocate sufficient bandwidth capacity between any logical port. If the TCAM capacity of a commodity switch is not sufficient to hold a full FIB computed by the routing stack, the network hypervisor can split the FIB over multiple physical switches (exactly as proposed in [2]).

Multi-tenant network architecture. In a multi-tenant network environment, the physical network is shared among tenants so isolation between tenant networks is a strict requirement. While in modestly sized networks this is rather easily achievable with today's solutions (such as VLANs), these solutions are unworkable at the scale already seen in production multi-tenant networks. For instance, the number of available VLANs is rather limited and, as the number of tenant networks grows, the management of the required configuration state becomes extremely brittle.[4]

The network hypervisor is perfectly suited to the multi-tenant challenge; it is sufficient for the hypervisor to allocate a logical switch per tenant and then map sufficient resources per logical switch. The isolation is taken care of by the mappings between physical and logical contexts. In this case, the logical context just happens to correspond to the tenants.

Tenants may be provided with full (self-service) control over their dedicated logical switches, freeing further resources from the physical network management. For example, the tenants can modify per (logical) port ACLs and they can even see (logical) statistics for their traffic. Similarly, any constraints due to integration with external networks can be represented using logical abstractions within the logical network view the tenants are provided with. For IP connectivity, a logical router may represent the IP subnet allocated for the tenant. Then it is the network hypervisor's task to provide such IP connectivity for the logical network by appropriately interconnecting the physical switches to the IP connectivity.

6 Related Work

As we have already discussed, networking makes heavy use of virtualization concepts at the lower levels with tunnels and tags (VLANs or MPLS), and multiple proposals have sought to virtualize higher-level interfaces to the network (examples include [4,9]). In our proposal, we argue that the correct layer at which to virtualize is the full forwarding plane, somewhere between these two extremes.

[3] We are currently investigating more sophisticated rule placement, as in [17].

[4] Q-in-Q (stacked VLANs) is a potential solution for the limited number of VLANs but its use requires even more planning than plain VLANs.

We now describe some recent proposals which relate to ours while acknowledging that additionally there is a vast body of prior work on which these ideas are built.

VROOM [16] proposes the use of a hypervisor between the control and forwarding planes of a router to facilitate migration. In our model, this is akin to running single logical forwarding element per physical switch. Our work extends this idea in two ways. First, we extend the logical view across multiple physical elements; second, we support multiple logical contexts sharing the same underlying hardware. Further, our work focuses on how such an approach can take advantage of traditional hardware forwarding paths to implement full forwarding plane virtualization.

Keller and Rexford (in [10]) have argued that the typical approach of overlaying a virtual network of multiple virtual routers on top of a shared physical infrastructure should be replaced by the "platform as a service" model of virtualization. With this we completely agree, and our focus here is on articulating the architecture necessary to make this a reality.

Overlay networks (*e.g.*, [1]) build networks on top of other networks for a variety of purposes. Here, we are proposing to use a fully logical network abstraction to express the desired functionality, and then use the network hypervisor to map this logical abstraction to the underlying hardware. In the process of mapping the logical to the physical, the network hypervisor effectively creates an overlay network, but the key distinguishing factor is the presence of the logical abstraction.

Various proposals (such as [3, 14, 15], and many others we omit for brevity) "slice" a physical network amongst multiple control plane instances. While these approaches are often described as "network virtualization", but in the broader systems literature, virtualization refers to the act of decoupling the (logical) service from its (physical) realization. In particular, the virtualized service may be implemented by a single physical component shared by multiple virtualized services *or* by using multiple physical resources to implement a single logical service.

The network hypervisor closely matches with the classic definition of the virtualization as it can both partition the physical network and build logical forwarding elements exceeding the capacity of any physical forwarding element. Accordingly, our proposal focuses on the problem of partitioning a single resource through the use of multiple contexts, *as well as* distributing the logical context over multiple physical elements. In contrast, slicing focuses on the former problem, by partitioning the physical forwarding space either through consuming physical ports, or partitioning address or tag space. Slicing does not provide the means for distributing the logical state across multiple physical elements, a key component of our design.

7 Discussion

The approach proposed here is conceptually quite simple: rather than requiring control planes to deal with the complicated and dynamic nature of physical networks, the network hypervisor allows them to specify the desired behavior in terms of a simple logical abstraction that is completely under their control. This logical abstraction is *only* as complicated as needed to express the desired behavior, and completely shields the control plane from the irrelevant details of the underlying physical infrastructure. The network hypervisor then assumes the responsibility of implementing this abstraction on the underlying physical network.

This approach merits further investigation, as we have only begun to scratch the surface of this idea.

8 References

[1] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. SOSP*, October 2001.

[2] H. Ballani, P. Francis, T. Cao, and J. Wang. Making Routers Last Longer with ViAggre. In *Proc. NSDI*, Apr 2009.

[3] S. Bhatia et al. Trellis: A Platform for Building Flexible, Fast Virtual Networks on Commodity Hardware. In *Proc. CoNEXT*, December 2008.

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

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

[6] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM CCR*, 35(5), 2005.

[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. In *SIGCOMM CCR*, July 2008.

[8] J. Hamilton. Data center networks are in my way. *Talk at Stanford Clean Slate CTO Summit*, 2009.

[9] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a Distributed Firewall. In *Proc. CCS*, 2000.

[10] E. Keller and J. Rexford. The "Platform as a Service" Model for Networking. In *Proc. INM WREN*, 2010.

[11] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. OSDI*, October 2010.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38(2), 2008.

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

[14] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *Proc. OSDI*, October 2010.

[15] J. S. Turner. A Proposed Architecture for the GENI Backbone Platform. In *Proc. of ANCS*, December 2006.

[16] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual Routers on the Move: Live Router Migration as a Network-management Primitive. In *Proc. SIGCOMM*, August 2008.

[17] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In *Proc. SIGCOMM*, August 2010.