RC25378 (WAT1304-079) April 26, 2013
Computer Science

IBM Research Report

VirtualWires for Live Migrating Virtual Networks across Clouds

Dan Williams, Hani Jamjoom
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

Zhefu Jiang, Hakim Weatherspoon
Cornell University
Ithaca, NY

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

VirtualWires for Live Migrating Virtual Networks Across Clouds

Dan Williams[†], Hani Jamjoom[†], Zhefu Jiang[‡], and Hakim Weatherspoon[‡]

[†]IBM T. J. Watson Research Center, Yorktown Heights, NY
[‡]Cornell University, Ithaca, NY

Abstract

Despite significant advances in enabling live virtual machine (VM) migration within a virtualized—cloud—infrastructure, *cross-cloud* live migration remains an ad hoc, complex process. To create a network environment in which live migration can occur, clouds are beginning to expose virtual networks as a service. Connecting, managing and maintaining membership and routing information for—possibly incompatible—virtual networks as VMs migrate between clouds is non-trivial for both cloud providers and cloud users. In this paper, we present *VirtualWire*, a system in which cloud providers offer—instead of a virtual network abstraction—a *connect/disconnect* primitive that is much easier to manage. VirtualWire offers cloud users a consistent method to create complex logical network topologies in which the virtual network fabric itself is able to be elastically stretched or live migrated within or between clouds. Leveraging nested virtualization, we have implemented and evaluated VirtualWire across third-party clouds, including Amazon EC2, achieving cross-cloud live migration of VMs and network components with as low as 1.4 s of downtime.

1 Introduction

Live migration has become an integral part of cloud infrastructure design. To offer live migration *as a service*, cloud providers[1] must enable VMs from other clouds to live migrate to their data centers. Live migration is a hypervisor-level operation, requiring coordination between the sending and receiving host machines. Despite recent advances in enabling hypervisor-level control across third-party clouds [36], networking remains a challenge for cross-cloud live migration.

Current live migration techniques are limited within a layer 2 network in order to prevent VM IP addresses from changing. Inside a single autonomous cloud data center, network architectures (e.g., NetLord [25], VL2 [15], and PortLand [26]) achieve scalable "virtual" layer 2 networks that enable flexible intra-cloud migration. Across clouds, providers have begun to expose virtual network abstractions that can seamlessly connect to private data centers. For example, Amazon's Virtual Private Cloud (VPC) [1] lets users bring their private or public IP addresses to the cloud using an IPsec tunnel.

Unfortunately, the focus on providing a virtual network abstraction does not automatically enable cross-cloud live migration. Today, cloud providers must manage the associated control logic that specifies how packets are routed within a virtual network. Providers, for example, may implement a distributed virtual switch or embed control logic into a software defined network (SDN) controller. Even emerging virtual networking wire format standards like VXLAN [23], STT [12], and NVGRE [30] assume the existence of a provider-managed control plane. In all cases, providers must also track to which—of potentially many—virtual network(s) each VM belongs. When VMs are migrated, the corresponding virtual networks must be appropriately stretched or reconfigured.

Furthermore, for cross-cloud deployments, this provider complexity does not eliminate the cloud user's task of network management. Especially during migration—both live and offline—cloud users must keep track of their virtual networks so that they can appropriately stretch them into a new cloud. The number of virtual networks that a cloud user must manage can be large. An enterprise application, for example, often uses multiple VLANs to isolate its various tiers. Moreover, each cloud may expose a slightly different network abstrac-

[1]We distinguish between cloud *providers* and *users*. Cloud providers manage the cloud, virtualized data center (e.g., Amazon Elastic Compute Cloud and Rackspace). Cloud providers also represent infrastructure administrators in large private clouds. Cloud users are the consumers of the cloud infrastructure in the form of leased VMs. They do not have hypervisor or root privileges to the physical infrastructure that powers the cloud.

tion, which may lack consistent support for important network protocols or middlebox configurations [9, 17].

Instead of a virtual network abstraction, we propose that providers expose a *connect*/*disconnect* primitive from which cloud users can build virtual networks. To this end, we present VirtualWire. VirtualWire is a system that couples every virtual network interface (vNIC) with a point-to-point network tunnel. vNICs belong to VMs that either implement servers or user-level network components (e.g., routers, switches, middleboxes) similar to VIOLIN [19]. Cloud users can construct complex virtual networks by connecting pairs of vNICs together. Connections between vNICs are maintained even as VMs migrate, eliminating any need for reconfiguration inside guest VMs.

VirtualWire has two main advantages over provider-supplied virtual network abstractions from a cross-cloud migration perspective:

- **Simplicity for Providers.** With VirtualWire, the cloud provider's task is reduced from managing the virtual networks' control plane to ensuring efficient delivery of network traffic across paired virtual interfaces (as specified by the user). More importantly, it enables decentralized management of vNICs. As part of the VM live migration process, the participating hosts (i.e., the sending and receiving hypervisors) only need to adjust the affected pair of vNIC tunnel destinations, effectively stretching the tunnel.
- **Consistency for Users.** VirtualWire enables users to completely define their virtual network. The cloud user controls the protocols supported by the virtual network and how traffic flows through middleboxes. The user specifies the peerings between different vNICs of virtual network components—once—in a process that mimics the act of plugging networking cables into network interface cards. The physical locations of all network components and the details of the underlying physical network are irrelevant to the user; network components or servers can freely migrate without reconfiguration from the user.

To date, we have implemented VirtualWire in Xen [6] to transparently intercept packets from virtual network interfaces and tunnel them across the physical network. Leveraging the Xen-Blanket [36], we have deployed VirtualWire on Amazon EC2 (a third-party cloud) as well as a private enterprise cloud environment. We have performed live VM migration from our private cloud to Amazon EC2—with no changes to VMs or the network topology—incurring as low as 1.4 s of downtime. We also demonstrate that VirtualWire is flexible enough to support a 3-tier application with rigid network address constraints between multiple firewall middleboxes onto EC2, achieving performance within 9%[2] of a native EC2 deployment.

In summary, VirtualWire makes three main contributions:

- a connect/disconnect primitive to enable virtual networks within and across clouds without requiring providers to manage a virtual network abstraction,
- an application of nested virtualization to extend the user's virtual network across one or more third-party clouds, and
- a demonstration of virtual network elasticity through live VM migrations of virtual servers and network components between our local setup and Amazon EC2.

The rest of this paper is organized as follows. Section 2 provides context with respect to related work. Section 3 and Section 4 describe the design and implementation of VirtualWire, respectively. Section 5 evaluates VirtualWire, including live VM migration between clouds; Section 6 concludes.

2 Background and Related Work

Live migration [11, 27] is a common hypervisor-level feature within a single virtualized environment. Live migration *as a service* requires the provider to expose an interface in which hypervisors can transfer VMs to other hypervisors on potentially different clouds. Using nested virtualization [8, 36], such an interface can be exposed on today's clouds. However, today's live migration strategies have stringent networking requirements. To enable a VM to maintain its IP address, live migration is limited to a layer 2 network. In the context of related work, we highlight requirements for enabling cross-cloud live migration that drive the design of VirtualWire.

Data Center Architectures. Several virtual networking architectures are built with VM migration in mind. VL2 [15] and NetLord [25] create a virtual layer 2 network abstraction that can scale to hundreds of thousands of VMs, partially motivated by the perceived need for flexibility in VM migration and assignment. Software defined networking (SDN) [16, 22, 24, 28] is emerging, in which a network-wide control platform can enable networks to adapt to the new challenges presented by live migration. However, these architectures have focused on a single provider or administrative domain. From a cross-cloud perspective, the underlying provider complexity in managing the virtual network does not reduce

[2] The overhead is introduced by the nested virtualization layer, which was needed to enable hypervisor level controls on EC2.

the complexity perceived by the user, who must manage a virtual network across—potentially inconsistent—abstractions. *VirtualWire should simplify cloud providers while offering users a consistent networking interface, despite migration across clouds.*

Stretching Across Clouds. Virtual Private Networks (VPNs) are commonly used to extend private networks into remote sites. CloudNet [37] relies on provider based VPNs to connect data centers and implements optimizations to migrate VMs across the WAN. Similarly, Amazon VPC [1] allows users to pick their own IP addresses within a subnet connected to a private network via a VPN. VNET [32] examines extending a layer 2 network to a remote environment by using tunnels to a proxy. More recently, vCider [4] and VPN-Cubed [5] use similar mechanisms to support layer 2 protocols in the cloud and even provide some control over the network topology, but require configuration in the guest operating systems. CloudSwitch [2] operates in an isolation layer that avoids guest operating system configuration, but does not facilitate the implementation of flow policies in the cloud. Unfortunately, the lack of support for networking features in the cloud, such as enforcing VLANs or middlebox interposition, continues to present significant migration challenges [9, 17]. Connecting clouds is not enough. *VirtualWire should support complex topologies and low-level protocols across clouds.*

Virtual Network Components in an Overlay. In order to support complex topologies and low-level protocols, overlays of virtual network components, such as VIOLIN [19], have been proposed. VINI [7] applies some of the ideas of VIOLIN in a controlled, realistic setting to create a shared network testbed. Emulab [35] also allows users to specify arbitrary network topologies that are emulated in software. In these systems, the overlay is largely static and not designed for live migration. Migration is disconnected from the management of overlays and therefore constrained by how the overlay is configured. *VirtualWire should integrate migration into a virtual network so that the network can automatically stretch to anywhere a VM migrates.*

Migrating Virtual Networks. Especially for large, multi-tier deployments (e.g., enterprise applications), live migration of network components with VMs as an ensemble is necessary [20]. To this end, LIME leverages SDN techniques and requires infrastructure providers to implement support for migration in the SDN controller. Individual virtual network components, particularly routers, have been migrated in VROOM [34]. More generally, a case has been made for network infrastructure providers to be completely decoupled from service providers [14]. However, these architectures focus on a single cloud provider. *VirtualWire should provide support for the live migration of virtual network components across clouds without needing explicit coordination between providers.*

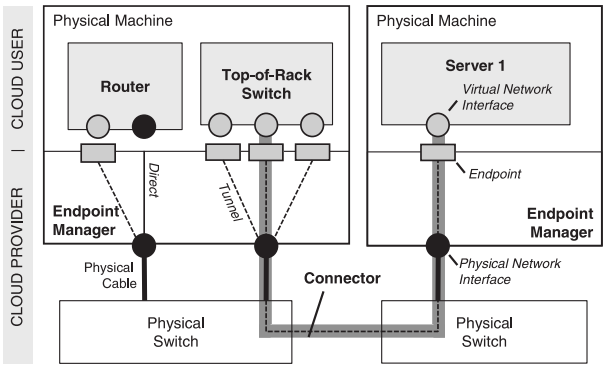


Figure 1: VirtualWire

3 VirtualWire

This section describes the high-level design of VirtualWire, and highlights how providing a connect/disconnect primitive from which users can implement their own virtual network results in low management overhead. We also discuss live migration and challenges in building an efficient virtual network on VirtualWire.

A virtual network in VirtualWire has a split architecture by design, illustrated in Figure 1. The cloud user is responsible for configuring the *user layer*, made up of *virtual network components*, while using a simple API to attach *connectors* between the virtual network interfaces (vNICs) on components. Connecters are optimized network tunnels that maintain the virtual network topology regardless of where virtual network components and servers are located; the provider implements the connector abstraction in the *provider layer*.

3.1 User Layer: Network Components

Similar to VIOLIN [19], in VirtualWire, users run and configure *virtual network components*, which are VMs containing configurable software implementations of switches, routers, and middleboxes. Many software implementations of network components exist (e.g., Open vSwitch [28], Click router [21], and XORP [18]). Even commercial vendors have started providing software equivalents of their physical network components. For example, Cisco has released the Nexus 1000V series of production virtual switches [10]. Although not production, NetSim [3] contains software implementations of 42 routers and 6 switches. Similarly, the Olive JUNOS implementation for training on Juniper devices runs on FreeBSD.

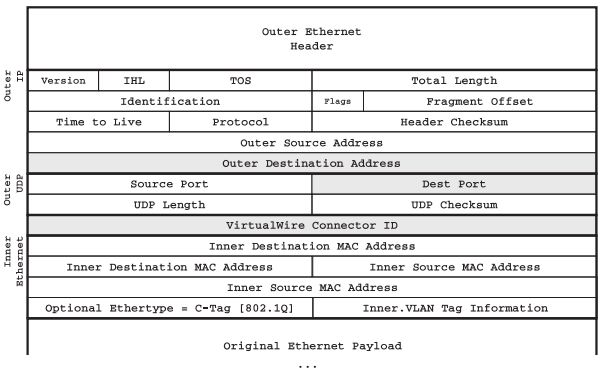


Figure 2: VirtualWire encapsulation

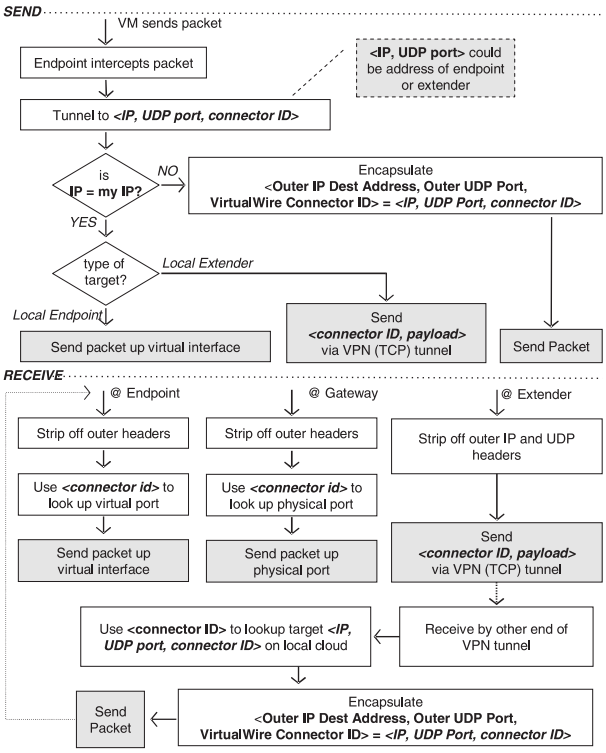


Figure 3: Path of a packet in VirtualWire

The use of virtual network components ensures that a cloud user can consistently implement even complex network semantics without provider support. From the provider's perspective, the combination of these components comprise a cloud user's virtual network in its entirety, simplifying migration and management. However, virtualized network components are limited by the performance of the underlying physical host and are unlikely to perform like physical network components.[3] Considerations when building efficient virtual networks in VirtualWire are discussed in Section 3.5.

3.2 Provider Layer: Connectors

The provider links together vNICs on virtual network components as specified by the user using *connectors*. Connectors have two *endpoints*, each bound to a vNIC. The binding between endpoint and vNIC is configured by an *endpoint manager* residing on a hypervisor. This binding does not change, even if a network component is migrated to another hypervisor. On migration, the configuration (e.g., tunneling destinations) of endpoints is updated to ensure the virtual network topology is maintained (see Section 3.4).

Encapsulation. Connectors are layer-2-in-layer-3 network tunnels. A layer 2 packet sent on a vNIC enters its associated endpoint, which encapsulates the full packet (with the entire MAC header, including VLAN tags) in a UDP packet; the 44 byte header is shown in Figure 2. The encapsulation used by endpoints is similar to the wire format in VXLAN [23]; however, unlike VXLAN, the header encodes a 32-bit connector ID instead of a network segment (e.g., a VNI in VXLAN terminology) because the provider does not need to maintain a notion of network segments. The target IP address and port number correspond to the physical network address of the target endpoint manager. Upon receipt of a packet, the endpoint manager strips the outer headers, examines the connector ID, and forwards the packet to the target endpoint. Figure 3 details the path of a packet in VirtualWire. In order to maintain the network topology, every endpoint sends packets to exactly one other endpoint. If an endpoint is migrated, the migration process ensures the relevant endpoint configuration is updated to address encapsulated packets to the correct endpoint manager (further discussed in Section 3.4).

The extra 44 bytes added to each packet introduces a risk of fragmentation. VirtualWire adopts the path MTU discovery technique used by NVGRE [30] to enable a server to deduce the reduced MTU and avoid fragmentation of packets after encapsulation. Since encapsulated packets are unreliable, it is possible that they may get dropped by the network for a number of reasons, including downtime during migration of a component or endpoint. This is akin to the unreliability of Ethernet, in which packets are delivered in best-effort fashion. If greater reliability is desired, a reliable transport protocol may be used to implement a connector.

Optimizations. Connectors between virtual network components running on the same hypervisor are auto-

[3] For example, a physical switch may have 32 ports, all capable of achieving 1 Gbps. An equivalent switch in VirtualWire may be hosted on a physical machine with a single 1 Gbps interface. The virtual switch will still have 32 ports, but is now only capable of achieving 1 Gbps—in aggregate—due to the limitations of the physical interface.

matically collapsed by the endpoint manager. In particular, the endpoint manager configures the endpoints to route packets directly to the co-located endpoint, rather than encapsulating self-addressed packets (see Figure 3). If, at a later time, one of the virtual network components is migrated to a different hypervisor, the endpoint manager re-enables encapsulation.

Extenders. Similarly, connectors also automatically extend across cloud networks and firewalls. Instead of addressing encapsulated packets directly to the recipient endpoint manager, packets are rerouted to an *extender*. An extender is a server (or set of servers) that both acts as an endpoint manager and maintains a permanent VPN tunnel to another cloud. Extenders are well known within a provider; any endpoint manager within a cloud knows of the extenders and how to use them to connect to endpoints on other clouds. As shown in Figure 3, an extender may be local (co-located with an endpoint). Packets arriving at an extender are automatically sent across the VPN tunnel, where they enter a new endpoint that encapsulates the packet and sends it to the target endpoint manager. From the point of view of the network components, the extender is invisible: the two network components remain logically connected. If the virtual network components migrate to the same cloud, the extender will no longer be used.

3.3 Connector Management

VirtualWire exposes a simple interface made up of two operations, `connect` and `disconnect`, which create and destroy connectors between the vNICs on virtual servers and network components. Similarly, VirtualWire exposes a `connect` and `disconnect` for extenders. These interfaces augment the typical VM creation and destruction interfaces exposed by cloud providers. Management of endpoints and connectors is intrinsically decentralized; no additional configuration database is required.

A virtual network created with VirtualWire can be connected to an existing physical network using a physical network appliance called a *VirtualWire gateway*. A VirtualWire gateway is a server with one or more physical network interfaces and VirtualWire endpoints. From the point of view of the physical and virtual network components on either side of the gateway, the gateway is transparent: traffic sent down a network cable from one component will arrive at the next component on the same interface as if both were in the physical network.

3.4 Live Migration

As providers begin to support live migration *as a service*, they will expose APIs for their hypervisors to communicate with other hypervisors—perhaps on a different cloud—to initiate migration and copy memory and metadata between them.[4] For example, a cloud provider may enable a cloud user to migrate a VM by specifying a destination hypervisor address. VirtualWire augments this mechanism to maintain the logical network topology as the VM migrates.

In particular, the endpoint manager on each hypervisor is integrated with its live migration mechanism. The endpoint manager on the source hypervisor maintains a number of connector endpoints for the migrating VM, depending on the number of vNICs it has. During migration, the source copies the endpoint configurations to the destination as part of the VM metadata. By definition, the source can also communicate with the hypervisor(s) hosting the sister endpoint(s) (perhaps through one or more extenders). The source copies information about the destination to each so that every sister endpoints can be updated. If the sister endpoint is on a different cloud than the destination, both endpoints are configured to communicate with the appropriate extender.

3.5 Building an Efficient Virtual Network

Live migration creates challenges for maintaining efficiency in a virtual network, especially if the virtual network crosses cloud boundaries. Two conflicting design goals must be considered when building virtual networks on VirtualWire.

Splitting Components. It may be difficult to efficiently place shared network components to maximize co-location if servers are migrating. For example, consider a deployment containing a central switch between two communicating pairs of VMs. If one pair of VMs migrates to another cloud it may be more efficient to replace the switch with two switches and place one switch in each cloud. Alternate, split versions of a switch, such as a distributed virtual switch (e.g., VMware DVS [33]), can be implemented in VirtualWire by connecting each part of the DVS with VirtualWire connectors. Importantly, even if the user runs a distributed component, the provider only manages endpoints, not a control plane.

Reducing Device Sprawl. Long chains of components can cause convoluted and inefficient routing paths as components or servers are migrated. For example, a chain of switches may involve packets that originate at one cloud, travel to a switch in another cloud, only to return to the next switch in the chain in the original cloud. Such chains may be able to be collapsed into fewer components in VirtualWire. For instance, a chain or tree of switches may be able to be implemented as a single big switch. However, collapsing components is at odds

[4] Alternately, nested virtualization enables hypervisor-level control on third-party clouds [36] to expose such APIs today.

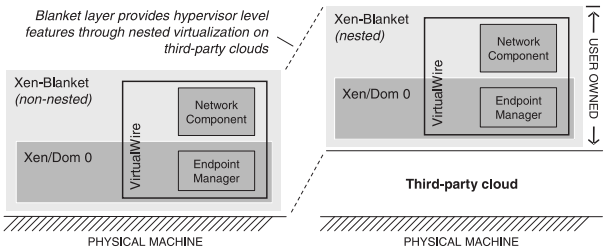


Figure 4: Nested Virtualization

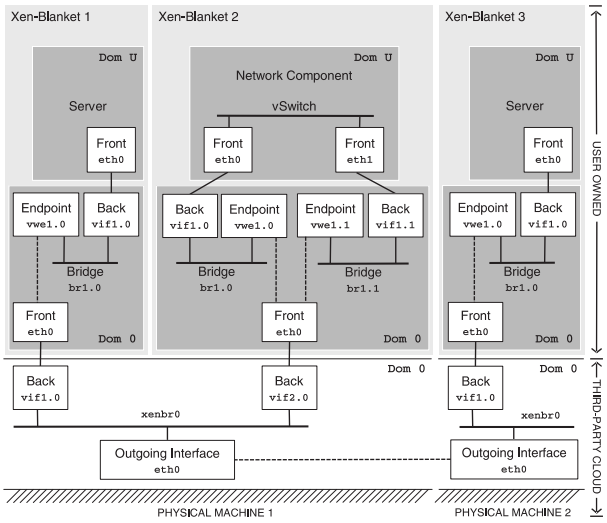


Figure 5: Implementation

with splitting components (described above). To navigate this tradeoff, ongoing work is considering live splitting and merging (collapsing) of virtual network components [29].

4 Implementation

We have implemented VirtualWire on an enterprise cloud, an academic testbed, and Amazon EC2. We augment existing hypervisor-level functions that provide the basic VM migration mechanisms for a live migration service. For Amazon EC2, where we do not have access to the underlying hypervisor, we leverage nested virtualization [8, 36] (Figure 4).

4.1 Nested Virtualization Background

We leverage the Xen-Blanket [36] to deploy VirtualWire across a mixture of public and private clouds. The Xen-Blanket is an open-source nested virtualization system consisting of a modified Xen [6] hypervisor that runs—without special support—on top of a variety of cloud providers or directly on hardware. The Xen-Blanket contains the Xen live migration implementation that cooperates with other Xen hypervisors to migrate a VM.

Guest VMs on the Xen-Blanket are assigned vNICs that use paravirtualized Xen split (front- and back-end) network drivers. Each guest vNIC is a front-end with a corresponding back-end in the control domain (Domain 0) of the Xen-Blanket. All guest VM packets pass through the back-end vNIC.

4.2 Network Components

We implement network components in Xen-Blanket guest VMs. For example, Figure 5 shows two servers connected to a switch component. The switch component is implemented by running the standard Linux software bridge in a Xen-Blanket guest VM configured with two vNICs. To run a server or component VM on any Xen-Blanket hypervisor on any cloud, we place virtual disk images containing the root filesystem of VMs in a globally accessible NFS share.[5] VMs on the Xen-Blanket can be live migrated using Xen's live migration mechanism [11]; we discuss enhancements to it in Section 4.5.

We relax the isolation of the VM for some network components, like the Linux software bridge. Instead, we implement the switch in Domain 0, as shown in Figure 6(a), and avoid two additional packet copies required to transfer a packet into and out of a guest VM. Because the switch is a simple component, migration is achieved by remotely reconfiguring Linux bridges with the `brctl` command. However, complex network components cannot be as easily implemented or migrated in Domain 0.

4.3 Connectors

Connector endpoints are bound to back-end guest vNICs using a dedicated software bridge in the Xen-Blanket Domain 0, as shown in Figure 5. Endpoint tunneling is performed via the endpoint manager kernel module. We implemented tunneling in a Domain 0 kernel module to reduce the number of context switches required when compared to a userspace implementation, such as `vtun`, which uses the TAP/TUN mechanism. A kernel implementation is especially important on the Xen-Blanket because nested virtualization magnifies the cost of context switches [36]. Inside the kernel module, each endpoint has an associated socket and thread to listen for UDP packets received on the external interface, but destined for the endpoint. Outgoing packets are intercepted and encapsulated in UDP packets as described in Section 3.2, then sent through Domain 0's external interface.

[5] Providing an efficient mechanism to access VM disk images from any cloud is out of the scope of this paper.

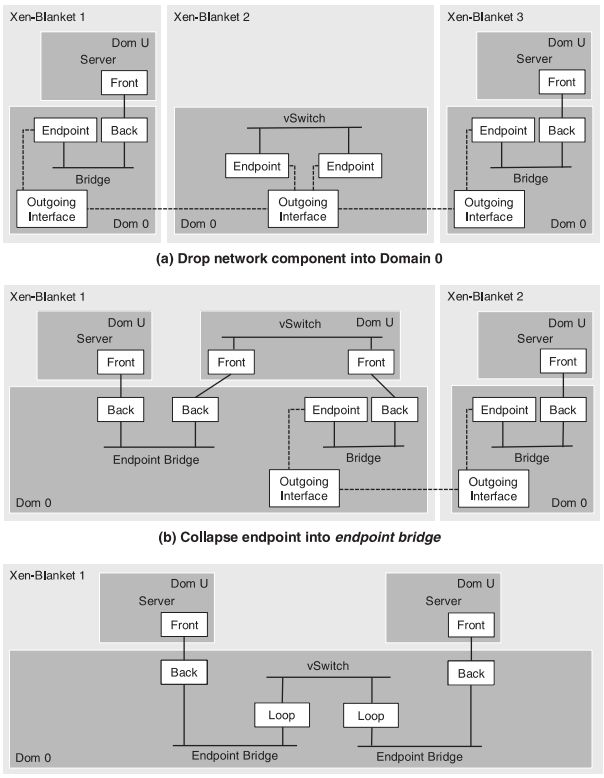


Figure 6: Implementation optimizations

If both endpoints of a connector are co-located on the same Xen-Blanket instance, the endpoint manager instructs the endpoints to forgo encapsulation. Instead, the back-end vNICs from the network components are connected through a software bridge in Domain 0, called an *endpoint bridge*. The configuration after collapsing one endpoint is depicted in Figure 6(b). Endpoints are collapsed even when a virtual network component is implemented in Domain 0. Figure 6(c) shows the configuration if all three VMs are co-located and the switch is implemented in Domain 0.

Extenders are implemented with OpenVPN. Each Xen-Blanket instance is configured with the location of all extenders from the current cloud to all other clouds. To improve performance, the extender is implemented in Domain 0.

4.4 Connector Management

We implemented an interface in the `/proc` filesystem in the endpoint manager kernel module, shown in Figure 7, through which endpoints are controlled. This interface provides a convenient mechanism to create and destroy them as needed. It is also used by the local migration process to update endpoints during a VM migration (Section 4.5). A VirtualWire Gateway is implemented as a server with an interface to a physical network and one or more connector endpoints.

```
/* create a new endpoint N on instance A that receives
   packets on Adr_A and sends packets to Adr_B */
echo "<N> <Adr_A> <Adr_B>" > /proc/vw/create
/* return current endpoint config */
cat /proc/vw/<N>
/* update endpoint send config (e.g. other end migrated) */
echo "<Adr_C>" > /proc/vw/<N>
/* destroy the endpoint (e.g. this end migrated) */
echo "<N>" > /proc/vw/destroy
```

Figure 7: The proc interface to control VirtualWire endpoints.

4.5 Live Migration

We modified the local Xen live VM migration process [11] to include endpoint migration. During live migration, the source hypervisor instructs the destination hypervisor to create an empty VM container. Then, the source iteratively copies memory to the destination, while the migrating VM continues to run. At some point, the source stops the VM and copies the final memory pages and other execution state to the destination.

In VirtualWire, we modify this process in two ways. First, when the destination creates the empty VM container (during the pre-copy phase), it also creates a set of endpoints that will encapsulate outgoing packets with the correct VirtualWire header. It also creates a set of bridges to bind the back-end vNICs of the migrating VM to these newly created endpoints. The endpoint configurations are supplied by the source and contain information about the clouds where sister endpoints reside. If sister endpoints reside on different clouds, the endpoints are configured to route to an extender.

Second, during the stop-and-copy phase, the source endpoint manager contacts the sister endpoints for all migrating vNIC connectors. Once again, endpoint configurations (the address and cloud of the destination) are supplied by the source. Each sister endpoint is updated, using an extender if necessary. Finally, after migration completes, the source deletes the original endpoints.

Unlike live VM migration today, a migrating VM does not end up behind a different switch port from the perspective of the logical network. Thus, no unsolicited ARPs need to be generated.

5 Evaluation

In this section, we first demonstrate cross-cloud live migration of VMs and network components from our local

	Downtime	Duration
Non-nested	0.7 [0.4]	19.86 [0.2]
Nested	1.0 [0.5]	20.04 [0.1]
Nested w/ endpoints	1.3 [0.5]	20.13 [0.1]

Table 1: Mean [w/standard deviation] downtime (s) and total duration (s) for local live VM migration

	Downtime	Duration
Local to Local	1.3 [0.5]	20.13 [0.1]
EC2 to EC2	1.9 [0.3]	10.69 [0.6]
Local to EC2	2.8 [1.2]	162.25 [150.0]

Table 2: Mean [w/standard deviation] downtime (s) and total duration (s) for live VM migration

environment to Amazon EC2. Then, we show that VirtualWire is flexible by evaluating a 3-tier enterprise application with a complex network configuration on EC2. Finally, we describe microbenchmarks and the performance of VirtualWire under various optimizations.

5.1 Experimental Setup

To evaluate VirtualWire, we use resources at our local institution as well as from Amazon EC2.

Local Testbed. In our local environment, we use up to two physical machines connected by a 1 Gbps network, each containing two six-core 2.93 GHz processors, 24 GB of memory, and four 1 TB disks. Each machine runs Xen with 2 VCPUs and 4 GB of memory dedicated to Domain 0. On this underlying system, we use HVM instances configured with 22 VCPUs and 12 GB of memory to run Xen-Blanket and VirtualWire.

Amazon EC2. In Amazon EC2, we use up to six Cluster Compute Quadruple Extra Large instances, each with 23 GB memory, 33.5 EC2 Compute Units, 1690 GB of local instance storage, and a 10 Gigabit Ethernet between instances.

Both environments use the Xen-Blanket patches for nested virtualization. The Xen-Blanket Domain 0 is configured with 8 VCPUs and 4 GB of memory. All guests (DomUs) are paravirtualized instances configured with 4 VCPUs and 2 GB of memory, and can run either on a single layer of virtualization in the local environment or nested within a Xen-Blanket instance. An NFS server running at our local setup provides VM disk images; VMs on Amazon EC2 access the NFS server through a VPN tunnel. For most of the experiments, we use `netperf` to generate 1400 byte packets in UDP_STREAM and UDP_RR modes for throughput and latency measurements, respectively.

5.2 Cross-Cloud Live Migration

In this subsection, we evaluate the ability of components and servers in VirtualWire to undergo live VM migration while maintaining the virtual network topology both within a single cloud and between clouds. Because we are unaware of other systems that enable cross-provider live migration, we do not compare the performance of migration in VirtualWire to other systems. For these experiments, we migrate a VM that is continuously receiving `netperf` UDP throughput benchmark traffic from another (initially co-located) VM. The network topology includes a virtual switch, also co-located, between the two VMs and implemented in Domain 0 for performance. For each experiment, we report the average (and standard deviation) application downtime and the total time the migration operation took to complete from 6 identical runs. Application downtime is calculated by examining periodic output from the `netperf` benchmark.[6]

The performance time of live migration of a VM (receiving `netperf` UDP traffic) between two machines in our local setup is shown in Table 1. The VM and its `netperf` partner VM were both run on a single layer of virtualization (*non-nested*) or nested setup (*nested* and *nested w/ endpoints*) before and after the migration. To isolate the nesting overhead from endpoint migration overhead, VirtualWire connectors are not used between the two VMs in the *non-nested* and *nested* experiments; instead, the physical network is bridged. We find a 43% increase in downtime and an 18% increase in total duration due to nesting. The added task of migrating VirtualWire endpoints introduces an additional 30% increase in downtime, but a negligible increase in total duration.

Table 2 quantifies the performance of live migration across clouds while maintaining the virtual network topology using a VirtualWire connector. We compare the performance of single-cloud live migration within our local nested setup (*Local to Local*) and within Amazon EC2 (*EC2 to EC2*) to multi-cloud migration between the two (*Local to EC2*). Within one cloud, local or EC2, the latency between the instances is within 0.3 ms, whereas across clouds it is about 10 ms. VPN overhead limits throughput across clouds to approximately 230 Mbps. The 10 Gbps network between our EC2 instances leads to significantly reduced total migration time when compared to local; however, the downtime was comparable. Live migration of a VM (receiving `netperf` UDP traffic) between our local nested setup and Amazon EC2 has

[6]The frequency of the periodic output is set to 0.1 s, so we cannot measure downtime less than 0.1 s.

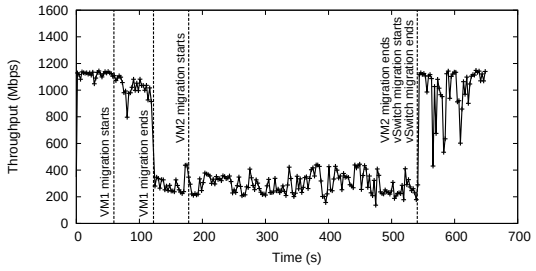


Figure 8: Throughput over time between two VMs migrating to Amazon EC2

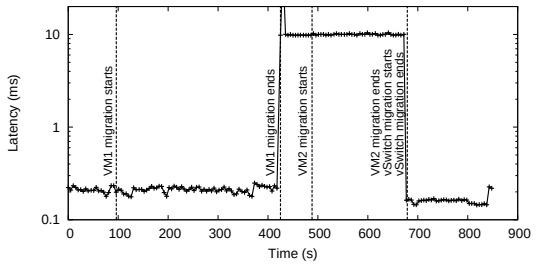


Figure 9: Latency over time between two VMs migrating to Amazon EC2.

a downtime of 2.8 s and a total duration of 162.25 s on average, but variation is high: the duration ranged from 48 s to 8 min. For an idle VM, the performance of the network between machines has little effect: the downtime during live migration between two local machines and from local to EC2 is 1.4 s on average.

We ran two more experiments,[7] shown in Figure 8 and Figure 9, to identify the throughput and latency over time for the test deployment as the recipient VM and then the switch and the sender VM were live migrated to Amazon EC2, respectively. Migration of the switch did not affect throughput because its Domain 0 implementation was trivial to migrate. As expected, significant degradation in the throughput and latency occurs when not all components are co-located on the same cloud.

5.3 Supporting Complex Networks

In this subsection, we show that VirtualWire can support a complex network topology of an application with multiple tiers and quantify the performance implications. As an application, we run the RUBiS [13] benchmark with three VMs representing the Web tier, application server tier, and database tier, respectively. To better represent the complexity that can arise in enterprise applications, we add two VMs running software firewalls (`iptables`) in between each tier. Like many real applications, the

[7] We could not measure both the throughput and latency from a single experiment using `netperf`.

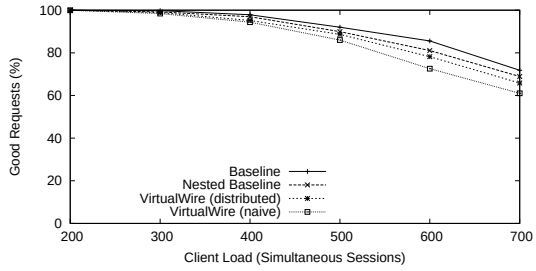


Figure 10: Performance of RUBiS application on EC2

VMs each have a hardcoded configuration—route table entries and IP addresses—that introduces constraints that necessitate a flexible virtual network.

Figure 10 shows the number of "good requests"[8] using the RUBiS application under various client load (simultaneous sessions). We examine four scenarios. The first two scenarios require manual modifications to the network configuration of each VM, but represent best-case scenarios in terms of application performance. The *baseline* is obtained by running each VM directly in an Amazon EC2 medium instance. The *nested baseline* runs an identically provisioned VM inside a Xen-Blanket instance, and therefore represents a best-case given nested virtualization overheads.

The other two scenarios are configurations of VirtualWire and require no modifications to the guest VMs. The first scenario, *VirtualWire naïve*, introduces a sixth VM that runs a virtual switch. Connectors are configured between each of the five VMs and the switch VM. The second scenario, *VirtualWire distributed*, replaces the switch VM with a distributed switch component as discussed in Section 3.5 and further evaluated in Section 5.6. In this configuration, VirtualWire achieves within 4% of the nested baseline and 9% of the baseline.

5.4 Microbenchmarks

Nesting Overhead. We isolate the overhead VirtualWire incurs due to Xen-Blanket nested virtualization using a `netperf` process in the Domain 0 of one physical machine across the 1 Gbps link in the local setup. We identify the throughput, latency and CPU utilization when network packets are received by a single-layer Domain 0 (`single dom0`), inside a guest running on the first layer (`single domU`), on the Xen-Blanket Domain 0 (`nested dom0`), or inside a guest running on the Xen-Blanket (`nested domU`). UDP throughput is maintained at line speed in all cases, while TCP throughput decreases by 6.7% for `nested domU`. However, as shown in Table 3,

[8] SPECweb2009, a similar benchmark, defines good requests as those which complete with a latency within 2000 ms [31].

	Latency	Agg. CPU Util.
Single Dom0	63 μs	61%
Single DomU	86 μs	88%
Nested Dom0	96 μs	97%
Nested DomU	210 μs	196%

Table 3: Network latency comparison in nested and non-nested virtualization settings

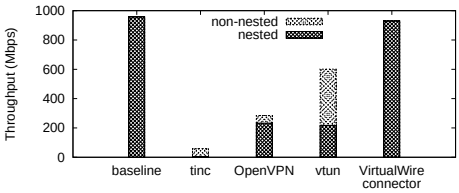


Figure 11: Tunnel throughput comparison

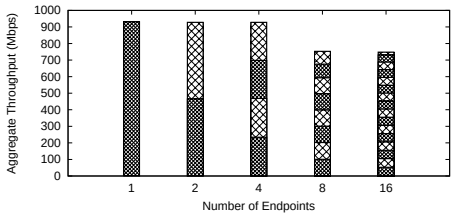


Figure 12: Scalability of VirtualWire endpoints in a nested Domain 0

	Connector w/o Extender		Connector w/ Extender	
	Throughput	Latency	Throughput	Latency
Local to Local	929.52 Mbps	.240 ms	233.00 Mbps	.300 ms
EC2 to EC2	1012.80 Mbps	.270 ms	230.83 Mbps	.310 ms
Local to EC2	n/a	n/a	239.96 Mbps	10 ms

Table 4: Performance of connectors with and without extenders

the latency increases sharply with a corresponding increase in CPU utilization due to extra packet copies and context switching.

Connectors. VirtualWire connectors are implemented in a kernel module and therefore avoid the high penalty for context switches experienced in the Xen-Blanket [36]. Figure 11 compares the performance of connectors with various software tunnel packages. On our local—both nested and non-nested—1 Gbps setup, we measure the UDP performance of a VM sending data to another VM through the tunnel. The baseline is a measurement of the direct (non-tunneled) link between the VMs. The kernel-module approach of VirtualWire connectors pays off: while connectors increase throughput by a factor of 1.55 over the popular open-source tunneling software `vtun` in a non-nested scenario, it achieves a factor of 3.28 improvement in a nested environment.

Endpoint Scalability. In VirtualWire, a network component with many ports, such as a switch or router corresponds to a DomU with many virtual network interfaces and many endpoints. To measure the scalability of endpoints, we increase the number of interfaces with endpoints in the DomU and run several simultaneous `netperf` UDP throughput tests over each connector. The results are shown in Figure 12. The 1 Gbps bandwidth becomes split between the individual connectors, but in aggregate remains high for at least 16 endpoints.

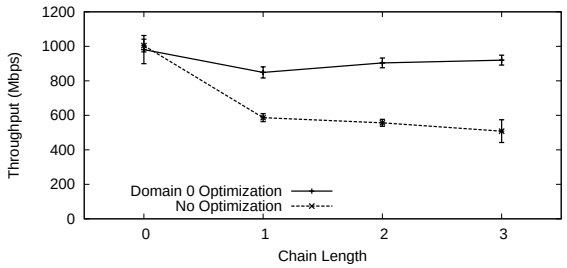
Extenders. VirtualWire uses OpenVPN to create extenders so that connectors can span clouds. Table 4 shows the performance (measured with `netperf`) between instances on our local nested setup or EC2 with and without extenders. As previously discussed (Figure 11), OpenVPN introduces significant overhead; when used as an extender, throughput and latency suffered by up to 25%. Across clouds, where extenders must be used, we also see high latency. This result underscores the importance of co-locating heavily communicating VMs on a single cloud, rather than across clouds when possible.

5.5 Component Chain Length

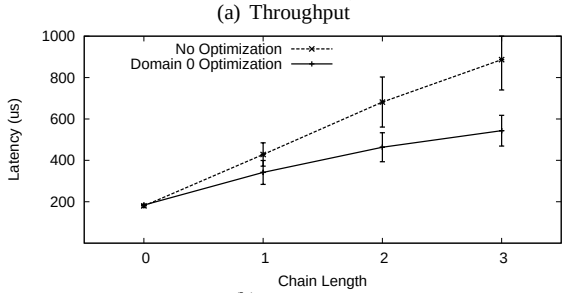
We examine the effects of chain-length within a VirtualWire network topology on throughput and latency with and without optimizations. We restrict our experiment to a single physical machine in the local environment to eliminate network congestion and jitter from the results. We vary the number of switches in the VirtualWire topology between the `netperf` endpoints. The switches are configured in a chain: each switch has two ports, while the server VMs have a single interface.

We first examine the case in which components are on different instances. We run five Xen-Blanket instances on a single physical machine, with 4 GB of memory and 4 VCPUs each, each hosting one VM. Figure 13(a) and Figure 13(b) show that, as expected, the throughput slightly decreases and the latency linearly increases as the chain length increases. By dropping all of the switch VMs into the Domain 0 of their respective Xen-Blanket instances, we found throughput was maintained with at least three switches on the path while latency was reduced from approximately 237 μs to 119 μs per switch.

If components share an instances, endpoints are collapsed and encapsulation is unnecessary. To evaluate this optimization, we run a single Xen-Blanket instance with 14 GB of memory and 22 VCPUs so that it can support up to five guests and Domain 0. Figure 14(a) and Figure 14(b) show the throughput and latency between the two servers with up to three switch VMs interposing on the packets in four configurations: either encapsu-

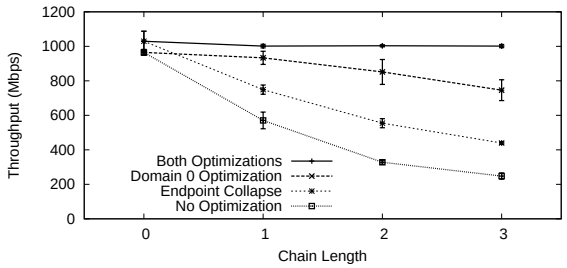


(a) Throughput

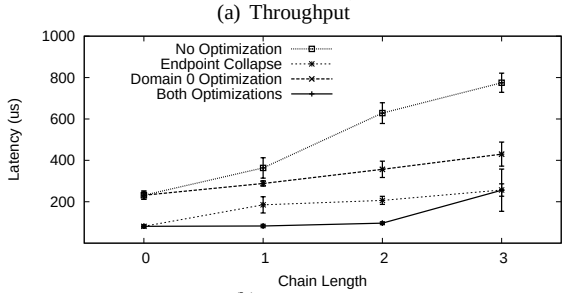


(b) Latency

Figure 13: Effects of component chaining on throughput and latency



(a) Throughput



(b) Latency

Figure 14: Effects of endpoint collapse and Domain 0 optimization in co-located component chains

lating or collapsing endpoints, each with DomU or Domain 0 switches. By avoiding encapsulation, endpoint collapse reduces latency by about 135 μs per switch. Combined with Domain 0 switches, good performance is maintained for all evaluated chain lengths. These results suggest that beyond co-locating heavily communicating components on the same cloud, VirtualWire derives significant benefit from co-locating components on the same instance.

5.6 Splitting Components

In this subsection, we demonstrate the performance advantages of splitting bottleneck network components. We run `netperf` between four server VMs on four EC2 Xen-Blanket instances. All VMs are connected to a virtual switch component running in Domain 0 on a fifth EC2 Xen-Blanket instance. Traversing the switch, each pair of VMs can achieve throughput of 1.39 Gbps on average. If two flows are active at the same time, throughput is split between the two flows, dropping them to 720.5 Mbps each.

We split the virtual switch component by by running a Linux bridge on each of the four EC2 Xen-Blanket instances and manually linking them (with connectors) and configuring them (with `ebtables`) to act as a single switch. The split version of the virtual switch avoids the bottleneck encountered with a centralized virtual switch component. In this case, each pair of VMs achieve throughput of 1.77 Gbps on average regardless of whether one or two flows are active.

6 Conclusion

Advances in virtual networking within clouds to support live migration do not directly apply across clouds. Despite the complexity incurred by providers to offer virtual network abstractions, cloud users are ultimately responsible for ensuring packets flow correctly through clouds and virtual networks. We have presented VirtualWire, a system that consistently enables expressive virtual networks across clouds while reducing provider involvement to the intrinsically decentralized task of managing pairs of vNICs. With VirtualWire, we have live migrated servers and network components to EC2, experiencing as low as 1.4 s of downtime. VirtualWire supports complex network topologies: a 3-tier application with address and middlebox requirements runs on EC2 without modification, maintaining within 9% of the performance of a native deployment. The inherent elasticity of VirtualWire—the ability to live migrate any network component in a virtual network—creates opportunities for placement optimizations and is well-suited to increasingly elastic cloud workloads.

References

[1] Amazon Virtual Private Cloud. `http://aws.amazon.com/vpc/`.

[2] CloudSwitch. `http://www.cloudswitch.com`.

[3] The Cisco Network Simulator & Router Simulator. `http://www.boson.com/netsim-cisco-network-simulator`.

[4] vCider. `http://www.vcider.com`.

[5] VPN-Cubed. `http://www.cohesiveft.com/vpncubed/`.

[6] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proc. of ACM SOSP* (Bolton Landing, NY, Oct. 2003).

[7] BAVIER, A., FEAMSTER, N., HUANG, M., PETERSON, L., AND REXFORD, J. In VINI veritas: realistic and controlled network experimentation. In *Proc. of ACM SIGCOMM* (Pisa, Italy, Sept. 2006).

[8] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The turtles project: Design and implementation of nested virtualization. In *Proc. of USENIX OSDI* (Vancouver, BC, Canada, Oct. 2010).

[9] BENSON, T., AKELLA, A., SHAIKH, A., AND SAHU, S. CloudNaaS: a cloud networking platform for enterprise applications. In *Proc. of ACM SoCC* (Cascais, Portugal, Oct. 2011).

[10] CISCO SYSTEMS, INC. Cisco nexus 1000v series switches data sheet. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9902/data_sheet_c78-492971.html`, Apr. 2012.

[11] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proc. of USENIX NSDI* (May 2005).

[12] DAVIE, B., AND GROSS, J. A stateless transport tunneling protocol for network virtualization (STT). `http://tools.ietf.org/html/draft-davie-stt-02`, Aug. 2012.

[13] EMMANUEL CECCHET AND JULIE MARGUERITE AND WILLY ZWAENEPOEL. Performance and Scalability of EJB Applications. In *Proc. of OOPSLA* (Seattle, WA, Nov. 2002).

[14] FEAMSTER, N., GAO, L., AND REXFORD, J. How to lease the internet in your spare time. *ACM SIGCOMM CCR 37*, 1 (Jan. 2007).

[15] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM* (Spain, Aug. 2009).

[16] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. *ACM SIGCOMM CCR 38*, 3 (July 2008).

[17] HAJJAT, M., SUN, X., SUNG, Y.-W. E., MALTZ, D. A., RAO, S., SRIPANIDKULCHAI, K., AND TAWARMALANI, M. Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud. In *Proc. of ACM SIGCOMM* (New Delhi, India, Aug. 2010).

[18] HANDLEY, M., KOHLER, E., GHOSH, A., HODSON, O., AND RADOSLAVOV, P. Designing Extensible IP Router Software. In *Proc. of USENIX NSDI* (Boston, MA, May 2005).

[19] JIANG, X., AND XU, D. Violin: virtual internetworking on overlay infrastructure. In *Proc. International conference on Parallel and Distributed Processing and Applications* (Hong Kong, China, Dec. 2004).

[20] KELLER, E., GHORBANI, S., CAESAR, M., AND REXFORD, J. Live migration of an entire network (and its hosts). In *Proc. of ACM HotNets* (Redmond, Washington, Oct. 2012).

[21] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, F. M. The Click Modular Router. *ACM Transactions on Compuer Systems 18* (August 2000), 263–297.

[22] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., , HAMA, T., AND SHENKER, S. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of USENIX OSDI* (Vancouver, Canada, Oct. 2010).

[23] MAHALINGAM, M., DUTT, D., DUDA, K., AGARWAL, P., KREEGER, L., SRIDHAR, T., BURSELL, M., AND WRIGHT, C. VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. `http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-00`, Aug. 2011.

[24] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR 38*, 2 (Apr. 2008).

[25] MUDIGONDA, J., YALAGANDULA, P., MOGUL, J., STIEKES, B., AND POUFFARY, Y. NetLord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters. In *Proc. of ACM SIGCOMM* (Toronto, Canada, Aug. 2011).

[26] MYSORE, R. N., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proc. of ACM SIGCOMM* (Aug. 2008).

[27] NELSON, M., LIM, B.-H., AND HUTCHINS, G. Fast Transparent Migration for Virtual Machines. In *Proc. of USENIX Annual Technical Conf.* (Anaheim, CA, Apr. 2005).

[28] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending Networking Into the Virtualization Layer. In *Proc. of ACM HotNets* (New York, NY, Oct. 2009).

[29] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/Merge: System support for elastic execution in virtual middleboxes (to appear). In *Proc. of USENIX NSDI* (Lombard, IL, Apr. 2013).

[30] SRIDHARAN, M., DUDA, K., GANGA, I., GREENBERG, A., LIN, G., PEARSON, M., THALER, P., TUMULURI, C., VENKATARAMIAH, N., AND WANG, Y. NVGRE: Network Virtualization using Generic Routing Encapsulation. `http://tools.ietf.org/html/-draft-sridharan-virtualization-nvgre-00`, Sept. 2011.

[31] STANDARD PERFORMANCE EVALUATION CORPORATION. Specweb2009 release 1.10 banking workload design document. `http://www.spec.org/web2009/docs/design/BankingDesign.html`, Apr. 2009.

[32] SUNDARARAJ, A. I., AND DINDA, P. A. Towards virtual networks for virtual machine grid computing. In *Proc. of Virtual Machine Research And Technology Symposium* (San Jose, California, May 2004).

[33] VMWARE, INC. vSphere Networking. `http://pubs.vmware.com/vsphere-50/topic/com.vmware.ICbase/PDF/vsphere-esxi-vcenter-server-50-networking-guide.pdf`, 2011.

[34] WANG, Y., KELLER, E., BISKEBORN, B., VAN DER MERWE, J., AND REXFORD, J. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. In *Proc. of ACM SIGCOMM* (Seattle, WA, Aug. 2008).

[35] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proc. of USENIX OSDI* (Boston, MA, Dec. 2002).

[36] WILLIAMS, D., JAMJOOM, H., AND WEATHERSPOON, H. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proc. of ACM EuroSys* (Bern, Switzerland, Apr. 2012).

[37] WOOD, T., RAMAKRISHNAN, K. K., SHENOY, P., AND VAN DER MERWE, J. CloudNet : Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Proc. of ACM VEE* (Newport Beach, CA, Mar. 2011).