

# Hosting Virtual Networks on Commodity Hardware

Sapan Bhatia\*, Murtaza Motiwala†, Wolfgang Mühlbauer‡, Vytautas Valancius†,  
Andy Bavier\*, Nick Feamster†, Larry Peterson\*, and Jennifer Rexford\*

\* Princeton University † Georgia Tech ‡ T-Labs/TU Berlin

## ABSTRACT

This paper describes Trellis, a software platform for hosting multiple virtual networks on shared commodity hardware. Trellis allows each virtual network to define its own topology, control protocols, and forwarding tables, which lowers the barrier for deploying custom services on an isolated, reconfigurable, and programmable network, while amortizing costs by sharing the physical infrastructure. Trellis synthesizes two container-based virtualization technologies, VServer and NetNS, as well as a new tunneling mechanism, EGRE, into a coherent platform that enables high-speed virtual networks. We describe the design and implementation, of Trellis, including kernel-level performance optimizations, and evaluate its supported packet-forwarding rates against other virtualization technologies. We are in the process of upgrading the VINI facility to use Trellis. We also plan to release Trellis as part of MyVINI, a standalone software distribution that allows researchers and application developers to deploy their own virtual network hosting platforms.

## 1. Introduction

Network services that have strikingly different requirements in terms of throughput, packet loss, security, or stability must nevertheless operate over a single, common communication infrastructure. Many of these services would benefit from having their own network topologies, and direct control over the routing, forwarding, and addressing mechanisms. For example:

- **Interactive applications** (*e.g.*, gaming, VoIP), which could run an *application-specific routing protocol* that converges more quickly than the existing network protocols that favor scalability over fast convergence.
- **Critical services**, which could run on a separate network with protocols tailored to defend against unwanted traffic (*e.g.*, denial-of-service attacks).
- **Enterprises**, which could construct and “lease” a private network connecting geographically disparate sites, with in-network support for key applications.
- **Network service providers**, which could run a separate “development” network for deploying and testing new configurations, protocols, and designs, and supporting “early-adopter” customers.

All of these needs could be addressed with the deployment of separate physical networks, each with customized protocols and topologies. Unfortunately, deploying physical infrastructure per service incurs tremendous space, power, and

management costs. Customizing today’s network devices is also challenging (and often impossible) because many support only limited, proprietary interfaces. These needs could also be addressed by deploying an overlay network that runs customized software on a distributed collection of computers connected to the Internet. However, overlays have limited visibility into, and control over, network conditions, and typically cannot forward traffic at high speeds.

Instead, we propose a “network hosting” platform that can run multiple programmable virtual networks over a shared physical network infrastructure. This hosting platform should have the following properties:

- **Speed:** A virtual network should be able to forward packets up to multi-Gigabit speeds.
- **Isolation:** To prevent virtual networks from interfering with one another, the infrastructure should support namespace and resource isolation of system resources (*e.g.*, process IDs, files, CPU) as well as network resources (*e.g.*, forwarding tables, link bandwidth).
- **Flexibility:** A service running inside the virtual network should be able to define its own routing protocol and application logic. The platform should provide a powerful and familiar development environment for network services.
- **Scalability:** The platform should be able to support many such virtual networks simultaneously to amortize its deployment and maintenance costs.
- **Low cost:** The cost for hosting a virtual network should be very low. Our hosting system should run on commodity hardware (*i.e.*, server-class PCs) to reduce costs and barriers to entry. Using commodity hardware also allows the infrastructure to more cheaply track advances in new technology (*e.g.*, multicore processors).

Our primary contribution is the design and implementation of *Trellis*, a platform for hosting virtual networks that achieves these goals. Trellis provides the substrate on top of which multiple fast and flexible virtual networks can run. Trellis synthesizes existing virtualization technologies (for virtualizing hosts and network stacks) with a new tunneling protocol and a new fast software bridge kernel module, to provide a scalable hosting platform with good isolation between virtual networks. The key challenge was to identify the right combination of technologies that could best satisfy our design goals: performance, scalability, isolation, and flexibility. We believe that the design choices and tradeoffs we have made hit a “sweet spot” along these axes for a virtual network hosting platform.

Our experiments demonstrate that Trellis is fast and scalable and provides good isolation between virtual networks. A virtual network hosted on Trellis can forward packets more than ten times faster than a similar overlay network [10]. Nodes running Trellis can host more than 60 virtual networks in parallel with no noticeable degradation in performance. The performance and jitter in any virtual network is nearly identical to that seen on native hosts.

We are deploying the Trellis software on our wide-area Virtual Network Infrastructure (VINI) facility [10]. The goal of VINI is to enable researchers to evaluate new protocols and deploy new services in an environment that is both realistic (*e.g.*, runs real routing software and carries real traffic) and controlled. We believe Trellis’s high performance and low jitter are a step towards achieving these goals. Because the need for low cost, isolated, reconfigurable, and programmable networks extends beyond the applications that will run on our modest-sized VINI facility, we also plan to release the Trellis software as part of MyVINI, a software distribution that allows researchers, network designers, and application developers to deploy their own virtual network hosting infrastructures. Beyond Trellis, MyVINI includes software for instantiating the virtual networks, including allocating system resources such as CPU and bandwidth.

The rest of the paper is organized as follows. Section 2 describes Trellis’s design, its two constituent components (*i.e.*, virtual hosts and virtual links), and their integration to support a virtual network. Section 3 assesses whether (and how) existing virtualization technologies, often designed for a different purpose, can support virtual hosts and virtual links in Trellis. Section 4 describes the implementation choices we made for Trellis to fulfill our design goals. Section 5 evaluates Trellis’s performance, scalability, and isolation relative to both native packet forwarding and other virtualization alternatives. Section 6 discusses our ongoing work, and Section 7 concludes.

## 2. Virtual Networks on Commodity Hardware

A *virtual network* is built using two components:

1. **virtual hosts**, which run software and forward packets, and
2. **virtual links**, which transport packets between virtual hosts.

Virtual networks constructed using commercial routers that support virtualization [24] currently have the limitation that they can only run one specific application (proprietary routing software and operating system) inside a virtual host. In contrast, Trellis is a virtual-network substrate that can run on commodity hardware using general-purpose operating systems. Trellis is likely to be cheaper to deploy than commercial offerings and can support a wide range of network services and applications. In this section we drive the design of Trellis from our requirements for (1) the entire system, and (2) the virtual hosts and virtual links that comprise it.

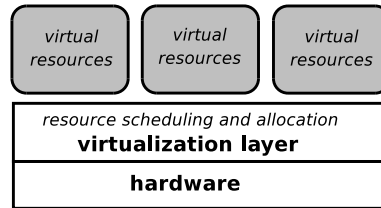


Figure 1: Overview of host virtualization, with virtual hosts shaded.

### 2.1 Trellis Design Requirements

We identify four high-level design requirements for Trellis. First and foremost, it must *connect virtual hosts with virtual links* to construct a virtual network. Second, it must run on *commodity hardware* (*i.e.*, server-class PCs) in order to keep deployment, expansion, and upgrade costs low. Third, it runs a *general-purpose operating system* inside the virtual hosts that can support existing routing software (*e.g.*, XORP [18] and Quagga [5]) as well as provide a convenient and familiar platform for developing new services. Finally, Trellis should support *packet forwarding inside the kernel* of the general-purpose OS, since forwarding every packet in user space introduces significant overhead and reduces the packet forwarding rate. An application running in userspace inside a virtual host can interact with devices representing the end-points of virtual links, and can write forwarding table entries (FTEs) to an in-kernel forwarding table (forwarding information base, or FIB) to control how the kernel forwards packets between the virtual links. Together, we believe that these design requirements place Trellis in the “sweet spot” mentioned earlier: it is the design point that best satisfies our overall goal of hosting fast, flexible virtual networks on a scalable and low-cost platform.

Understanding the design of Trellis first requires a better understanding of the requirements for both virtual hosts and virtual links. The next two sections define these components and detail their requirements.

### 2.2 Virtual Hosts

A *virtual host* sees the illusion of a dedicated physical host, even though multiple virtual hosts may be running on the same physical hardware. At a high level, a virtual host can be thought of as a “box” containing resources, as illustrated in Figure 1. A virtual host appears to have dedicated physical or logical resources inside this box; examples of physical resources are CPU, memory, and link bandwidth, whereas logical resources are resources implemented by the operating system such as the process table, page table, IPv4 forwarding table, memory buffers, etc. In reality, all of these resources are only “virtual” in that they are enabled by a virtualization layer that implements the virtual host abstraction. The virtualization layer creates virtual resources from physical ones using resource allocation and scheduling mechanisms, so that each virtual host receives its expected share of the resource in question. Likewise, the virtual host abstraction limits the scope of logical resources to inside the “box”, so that each virtual host can safely manipulate its own logical resources. The virtualization layer may not virtualize all possible resources, and so there may be resources resid-

ing on the physical machine but outside of the “box”; these resources may be either inaccessible, support limited interaction from within a virtual host, or be shared with other applications. Virtual hosts achieve two types of isolation:

- *Resource isolation* ensures that no virtual host can interfere with the resources (*e.g.*, CPU, memory, network bandwidth) that are allocated to another virtual host. Resource allocators in the virtualization layer multiplex and schedule these physical resources to provide virtualized resources inside a virtual host.
- *Namespace isolation* ensures that each virtual host can name and reference resources (*e.g.*, processes, files, memory, network interfaces, network addresses, forwarding tables) and cannot reference resources in other contexts. For example, an application in one virtual host is not able to add routes to the FIB of another virtual host, and two or more virtual hosts can use the same IP address to name different virtual interfaces.

In addition, Trellis’s virtual host technology must be *fast* and *scalable*. Our goal is to scale to approximately 50 active virtual hosts per node, in order to appropriately amortize the cost of the hosting platform (in our case, the VINI facility).

### 2.3 Virtual Links

A virtual link has the appearance of a physical link, but many virtual links may share a single physical link, and a virtual link may span many hops through the underlying physical network. In our design for Trellis, virtual links transport traffic between two virtual hosts. A virtual host transmits a packet on a virtual network interface to send it on a virtual link. After a packet exits the virtual host via the virtual interface, it is optionally rate-controlled by a traffic shaper (to enforce a maximum bitrate) before being tunneled to the other endpoints of the virtual link.

In Trellis, virtual links should support:

- *The appearance of a virtual ethernet.* We choose ethernet as the basis of Trellis’s virtual link abstraction because it is a ubiquitous and familiar layer-2 technology. A virtual link should provide ethernet semantics (*e.g.*, broadcast domains, point-to-multipoint topologies) and support the ethernet frame format.
- *Lightweight encapsulation and demultiplexing.* Because multiple virtual ethernet devices (and multiple virtual hosts) may share a single physical device, the substrate must ensure that packets are demultiplexed to the correct virtual interface (and virtual host).
- *Bandwidth enforcement outside the virtual host.* Each virtual link can have a bandwidth cap, to ensure isolation between virtual networks. The substrate should not permit the virtual link to send traffic in excess of this specified rate.

### 2.4 Trellis Design

Given these requirements, we now present the Trellis design. Figure 2 illustrates this design by showing a virtual network as hosted on Trellis. The functionality of the virtual network is spread across three layers: user space inside the

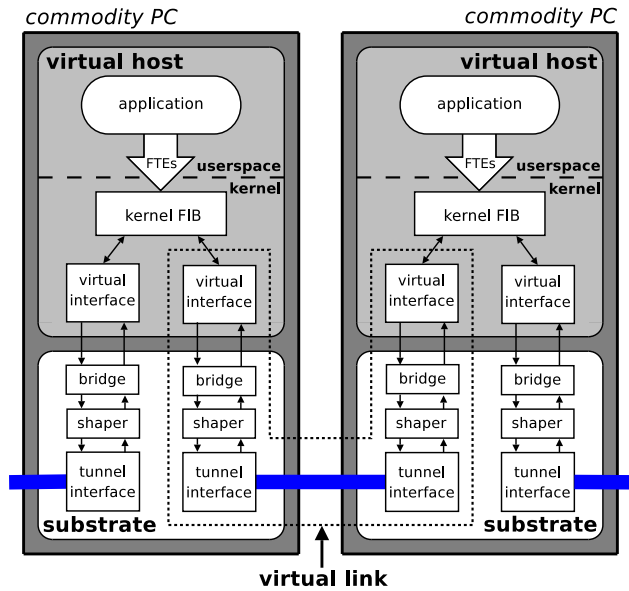


Figure 2: Overview of Trellis design, showing virtual hosts connected by a virtual link

virtual host; in the kernel inside the virtual host; and outside the virtual host in a substrate layer that is shared by all virtual networks residing on a single host. The elements inside a virtual host can be accessed and controlled by an application running on that virtual host. Elements in the substrate cannot be directly manipulated, but are configured by the Trellis management software on behalf of an individual virtual network. Of course, multiple virtual hosts can run on the same physical hardware, but this is not shown in the picture. Physical network interfaces are also not shown because they are hidden behind the tunnel abstraction.

The virtual links that connect virtual hosts in Trellis consist of four components, as outlined by the dotted U-shaped box in Figure 2: (1) two or more *virtual interfaces*, each with a unique MAC address; (2) a *tunnel* between the interfaces, incorporating both an underlying transport mechanism and a method for encapsulating and demultiplexing the packet; (3) a *bridge* that connects each virtual interface to a tunnel interface; and (4) a *traffic shaper* between each virtual interface and its corresponding tunnel interface. A virtual interface sends and receives packets for the virtual host. Once on a virtual link, the packet travels through a traffic shaper, and via a tunnel to the host on the other side of the virtual link. The remote host receives the packet, decapsulates it, and delivers it to the corresponding virtual interface, where it is received by the network stack in a virtual host. This virtual link presents to the host the appearance of a virtual point-to-multipoint ethernet link (*i.e.*, it is a link that will transport ethernet frames), although in practice this virtual link may span multiple hops in the underlying network.

We note several salient features of this design:

- *Per-virtual host virtual interfaces and tunnels.* Each virtual host is a node in a larger virtual network topology; thus, Trellis must be able to define interfaces and associated tunnels specific to that virtual network.

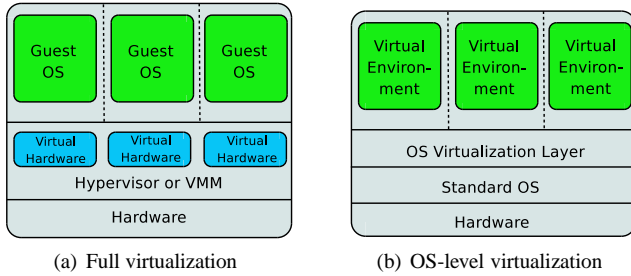


Figure 3: Two approaches for implementing virtual hosts.

- *In-kernel, per-virtual-host forwarding tables.* Each virtual host must be able to define how traffic is forwarded by writing its own forwarding-table entries. A virtual host’s forwarding table must be independent of other forwarding tables, and processes running on one virtual host must not be able to affect or control forwarding table entries on a different virtual host.
- *Separating virtual interfaces from tunnel interfaces.* Separating the virtual interface from the tunnel endpoint enables the creation of point-to-multipoint links (*i.e.*, the emulation of a broadcast medium). In addition, this separation allows the substrate to enforce a rate limit on each virtual link, to ensure resource isolation between the virtual networks.

The Trellis design incorporates virtualization, tunneling, packet demultiplexing, and traffic-shaping mechanisms to provide a substrate for hosting fast, flexible, and isolated virtual networks. Rather than implement Trellis from scratch, we chose to synthesize existing technologies into a working system. A key challenge in building Trellis was to identify and combine individual virtual host and virtual link technologies to provide the desired features. As it turns out, we ultimately implemented several new components because existing systems did not adequately meet our needs. The next section provides background on various technologies for host and link virtualization in order to motivate the specific implementation choices we made for Trellis. Section 4 describes the Trellis implementation.

### 3. Existing Virtualization Techniques

In this section, we discuss various approaches for virtualizing a host (full virtualization and “containers”) and a network (VLANs, VPNs, overlays, and logical routers). We summarize the strengths and weaknesses of each with regard to hosting virtual networks on commodity hardware.

#### 3.1 Virtual Hosts

A host can be virtualized using two mechanisms, as illustrated in Figure 3: *full virtualization*, whereby each virtual node runs its own instance of an operating system; and *OS-level virtualization*, whereby some of the operating system’s resources are isolated per-virtual host.

##### 3.1.1 Full virtualization

Full virtualization provides complete virtualization of the underlying hardware. As illustrated in Figure 3(a), the hard-

ware runs a Virtual Machine Monitor (VMM), also called a hypervisor, that hosts one or more “guest” operating systems. Each of the guests runs on a *virtual machine* (VM) provided by the VMM; all software that is capable of running on the underlying hardware can be run in the virtual machine itself. The VMM is responsible for implementing resource isolation among individual guests and the virtual machine abstraction naturally provides namespace isolation. A variant of full virtualization is called **paravirtualization**, which optimizes the hardware emulation to improve performance, but also requires modifications to the guest operating system. VMWare Server [39, 35] and Linux’s Kernel-based Virtual Machines (KVM) [1] provide full virtualization, while Xen [13] and Denali [41] are examples of systems which use the paravirtualization paradigm. Xen can also perform full virtualization on a CPU with virtualization support.

Running a separate operating system per virtual network can create unnecessary overhead for our network virtualization substrate. The guest OS runs more slowly than it would natively, because the trusted domain or VMM must intercept system calls and translate instructions that potentially interact with the native hardware (though this is somewhat alleviated in Xen, which allows guests to register a “fast” exception handler that bypasses the hypervisor). Full virtualization also requires copying data packets from the trusted domain or the VMM, which can degrade packet-forwarding performance considerably. Although recent work has attempted to improve the performance of virtual network devices in Xen [25], these optimizations are primarily aimed at bulk TCP transfers and only improve *throughput*, as opposed to the *packet-forwarding rate*.

##### 3.1.2 Network-based virtualization (“Containers”)

**Containers** (sometimes called “virtual environments”) partition operating system resources without requiring OS instances to be run in separate virtual machines as shown in Figure 3(b). A container-based operating system isolates some subset of the resources that it manages. The OS typically implements container virtualization using advanced scheduling techniques for physical resources (*e.g.*, CPU time), and tagging and contextualizing for logical ones (*e.g.*, kernel data structures). Multiple containers run on top of a single operating system kernel. Typically, fully virtualized and paravirtualized systems can provide better isolation than containers, but container-based systems have better performance since containers are more lightweight abstractions than virtual machines [33, 7].

Existing systems provide OS-level virtualization for various aspects of the operating system’s resources. Linux VServers [23], FreeBSD Jails [20], and Solaris Zones [36], add OS-level virtualization capabilities to the kernel; they securely partition OS resources, such as the file system and CPU time. The PlanetLab platform uses the Linux VServers for its OS-level virtualization [9]. Unfortunately, many of these technologies do not provide virtualization of the network stack, *i.e.*, they do not contextualize the variables in the network stack for each container. As a result, different containers share a common kernel forwarding table and, thus,

they cannot be used directly to allow each user to define a custom network topology or forwarding mechanisms.

A relatively new OS-level virtualization technology called OpenVZ [26], and its commercial counterpart Virtuozzo [38], allows virtualization of various OS-level resources, including the network stack. OpenVZ primarily aims to achieve efficient utilization of server resources (“virtual private servers”) and live migration of running applications; in contrast, we focus on how to use OS-level virtualization approach to construct virtual networks.

NetNS [11] is a prototype network stack virtualization technology that takes advantage of recently introduced virtualization APIs in Linux. NetNS does not virtualize an entire host, but rather provides each “network container” with its own in-kernel virtual devices, FIB, iptables settings, configuration variables, and so on. A process binds to a network container to obtain access to the virtual resources that it contains. One can think of NetNS as providing roughly equivalent functionality to OpenVZ’s network stack virtualization.

### 3.2 Virtual Networks

We describe the motivation behind existing technologies for building virtual networks and relate them to Trellis.

**Virtual Local Area Networks (VLANs)** [16] allow network operators to give hosts that are potentially topologically dispersed the appearance of being on an isolated LAN with a single broadcast domain and subnet. All frames bear a VLAN ID in the MAC header, and switches forward frames based on both the destination MAC address and the VLAN ID. Switched VLANs at different sites can be connected using *trunking* to tunnel VLAN-tagged frames between switches through the network. Assigning a set of hosts to the same VLAN offers many potential advantages, such as affording other hosts on the VLAN a higher level of trust, and being able to run broadcast protocols (*e.g.*, DHCP).

Trunked VLANs are examples of **virtual private networks (VPNs)**, which are virtual networks implemented by tunneling. Carriers may construct VPNs using technologies such as BGP/MPLS [31] or GRE to give customers the appearance of a dedicated network over a shared IP backbone. Today’s routers even provide some support for nested VPNs, such as Cisco’s Carrier Supporting Carrier [12], which allows one network to provide the MPLS backbone for another; and Inter-AS, which allows providers to “peer” to provide end-to-end VPN support. In contrast to router-supported VPNs, tools such as OpenVPN [8] enable construction of an isolated virtual layer-2/3 network between a set of edge hosts. This sort of host-based VPN does not require special support from the core network infrastructure.

A Trellis virtual link provides the same *abstraction* as a VLAN, and one could envision constructing Trellis using virtual hosts connected by trunked VLANs. This would require control over the switches and routers inside the network. Looking ahead to Section 4, the Trellis *implementation* tunnels ethernet frames over IP and so resembles a host-based VPN. This allows virtual networks hosted on Trellis to span multiple links and administrative domains without assuming administrative control over network devices.

**Overlay networks** compose networks from end systems and end-to-end paths; hosts are not typically virtualized, but each “link” in an overlay network comprises many IP-layer hops. Overlay networks treat the layer-3 network as a black box, and provide a way to improve end-to-end performance and reliability [6, 17] and deploy new distributed services [34, 29, 32]. In a sense, our previous work on PL-VINI [10] (*i.e.*, an initial prototype of network virtualization on top of the PlanetLab software) could be viewed as a particular instantiation of an overlay network that (1) is tailored to run software routers as a specific application and (2) allows multiple such “overlays” to run in parallel. As with conventional overlay networks, all forwarding in PL-VINI occurs in user space, and links are IP-level paths.

Recently, commercial router vendors have started supporting virtualization of their router hardware. **Logical routers** [24] decompose a single physical router into multiple logical routers that have their own routing tables, interfaces, policies, and routing-protocol instances. The primary driver for logical routers is consolidation of multiple network elements into a single hardware device, to simplify physical configuration (*e.g.*, racks and cables) and reduce space and power requirements. For example, an ISP can provide enterprise customers with access to logical routers (that the customers can configure), obviating the need to deploy separate physical edge routers. Support for logical routers also opens the door for running customized routing configurations, or even different routing protocols, for key applications.

Some of the motivations for Trellis are similar to commercial vendors’ reasons for supporting logical routers. In fact, Trellis can provide similar functionality by enabling multiple instances of routing software like XORP [19] and Quagga [5], to run in different virtual hosts on the same physical machine. However, we wish to provide a hosting platform with much greater flexibility and lower cost. A virtual network hosted on Trellis can run a much wider range of software than today’s IP routing protocols. We believe that supporting programmable virtual networks on a general-purpose operating system will lower the barrier for creating new control-plane protocols and network services.

## 4. Trellis Implementation

Trellis synthesizes host and network virtualization technologies into a single, coherent system that satisfies the design requirements in Section 2. In this section, we explain the implementation decisions we made when building Trellis to achieve our goals of speed, isolation, flexibility, and scalability.

### 4.1 Hosts: Container-Based Virtualization

**Decision 1** *Create virtual hosts using Container-based Virtualization (not full virtualization).*

As explained in Section 3, two common mechanisms for implementing virtual hosts are full virtualization and container-based virtualization (also sometimes called container-based operating systems, or simply “COS”). Our

requirements for good speed and scalability, and reasonable isolation and flexibility, suggest that container-based virtualization is more suitable for building Trellis. Therefore, we chose to synthesize two container-based approaches, Linux VServer [33] and NetNS [11], to serve as the virtual hosting environment of Trellis. Since the PlanetLab OS is also based on VServer, this allows us to leverage PlanetLab’s management software to run a Trellis-based platform. Another possible choice for a COS would have been OpenVZ; we evaluate both our approach and OpenVZ in Section 5.

Table 1 summarizes how full virtualization and COS compare with respect to speed, isolation, and flexibility. The rest of this section justifies our choice to use container-based virtualization in more detail.

**Speed.** Packet forwarding in Trellis must be fast; both full virtualization and forwarding in user space do not forward packets as fast as container-based virtualization. Previous studies have shown that, without optimizations, packet forwarding performance in Xen can suffer significant performance penalties due to packet multiplexing and demultiplexing overheads, the I/O channel between the driver domain and the guest domains, and bridging the physical interfaces to the back-end network interfaces [25].

Although some work has focused on optimizing performance in fully virtualized systems (*e.g.*, by optimizing memory usage and minimizing data copies in the guest OSes), a COS nevertheless provides faster forwarding, since packet forwarding can take place entirely within the kernel which avoids any data copying and scheduling overheads. Our results in Section 5 confirm these findings by comparing several container-based virtualization technologies to Xen.

**Isolation.** As previously mentioned, virtual networks must have both namespace isolation and resource isolation. Both full virtualization and container-based virtualization provide namespace isolation for many system resources, including the network stack. Full virtualization does provide more comprehensive isolation than container-based virtualization: for example, full virtualization protects against operating system crashes (*e.g.*, due to a buggy device driver or some other software fault). However, for the purpose of creating independent networks with independent resource allocations, both full virtualization and container-based virtualization provide a roughly equivalent amount of isolation: for example, both technologies prevent a virtual host from accessing the resources (*e.g.*, processes, files, network devices) of some other virtual host.

Hosts in virtual networks require the appearance of dedicated network interfaces: the behavior of each virtual link (*e.g.*, packet loss rate, latency, jitter) must *not* depend on the traffic patterns or load on other virtual networks that are sharing the physical infrastructure, which implies that the isolation provided by the virtual host must perform two functions: rate limiting and scheduling. At the moment, both full virtualization and container-based virtualization support traffic shaping in the root context. In principle, it is also possible for the root context to *schedule* the traffic on each virtual link to ensure that no virtual host sees inordinate de-

Criteria		Full Virtualization	COS
Speed	Packet forwarding	No	Yes
	Disk-bound operations	No	Yes
	CPU-bound operations	Yes	Yes
Isolation	Rate limiting	Yes	Yes
	Jitter/loss/latency control	Unknown	Yes
	Link scheduling	No	No
Flexibility	Custom data plane	Guest OS change	No
	Custom control plane	Yes	Yes

**Table 1: Container-based virtualization vs. full virtualization. Previous studies on container-based virtualization and full virtualization explain these results in more detail [33, 27].**

lays in sending or receiving traffic, though no such scheduling mechanisms yet exist for either full virtualization or container-based virtualization. Incorporating a scheduling mechanism for a fair allocation of resource across containers is an area for future work.

**Flexibility.** Virtual hosts in Trellis are connected by a virtual ethernet link; ethernet connectivity between hosts allows applications to run routing protocols between virtual hosts and have the appearance of a directly connected IP link. ethernet connectivity also allows different virtual hosts on the same physical host to number virtual interfaces from the same address space. Both container-based virtualization and full virtualization enable this function.

Virtual networks may wish to run custom control plane (*i.e.*, routing) software. For example, some virtual networks may wish to run a secure routing protocol (*e.g.*, S-BGP [21]), while others may not. Both types of virtualization provide sufficient isolation for this purpose. A thornier issue is custom data plane operations, such as forwarding non-IP packets, which requires modifications to the network stack in the operating system. In full virtualization, such customization is possible but requires modifications to the guest OS; unfortunately, container-based virtualization does not provide this flexibility because all virtual hosts share the same data structures in the kernel (recall that containers achieve separation by tagging according to context, not by allocating separate physical memory and data structures to each virtual host). We believe, however, that providing in-kernel data-plane customizability may be possible for container-based virtualization by partitioning kernel memory and data structures analogously to how similar systems have done this in hardware [37].

**Scalability.** Trellis should support a large number of networks running simultaneously. Previous work, as well as our experiments in Section 5, show that container-based virtualization scales better than other alternatives: specifically, given a fixed amount of physical resources, it can support more concurrent virtual hosts than full virtualization. This better scalability makes sense because in container-based virtualization only a subset of the operating system resources and functions are virtualized.

## 4.2 Links: Tunnels

**Decision 2** *Implement virtual links by sending ethernet frames over GRE tunnels (EGRE).*

Virtual links must be fast. First, the overhead of transporting a packet across a virtual link must be minimal when compared to that of transporting a packet across a “native” network link. Therefore, encapsulation and multiplexing operations must be efficient. Virtual links must also be flexible: they must allow multiple virtual hosts on the same network to use overlapping address space, and they must provide support for transporting non-IP packets.

We tackled these problems by implementing a new tunneling module for Linux, ethernet-over-GRE (EGRE). Trellis uses GRE [15] as the tunneling mechanism because it has a small, fixed encapsulation overhead and also uses a four-byte key to demultiplex packets to the right tunnel interface. This approach is much faster than approaches that perform a lookup on the source, destination address pair. Other user-space tunneling technologies like `vtun` [40] impose considerable performance penalty compared to tunnels implemented as kernel modules.

EGRE tunnels allow each virtual network to use overlapping IP address space, since hosts can multiplex packets based on an ethernet frame’s destination MAC address. This also allows Trellis to forward non-IP packets, which allows virtual networks to use alternate addressing schemes, in turn providing support for existing routing protocols that do not run over IP (e.g., IS-IS sometimes runs directly using layer 2 addresses). Currently, forwarding non-IP packets in requires running Click in user space, as in PL-VINI [10]. In our ongoing work, we are investigating how to implement virtualizable custom data planes; Section 6 discusses this problem in more detail.

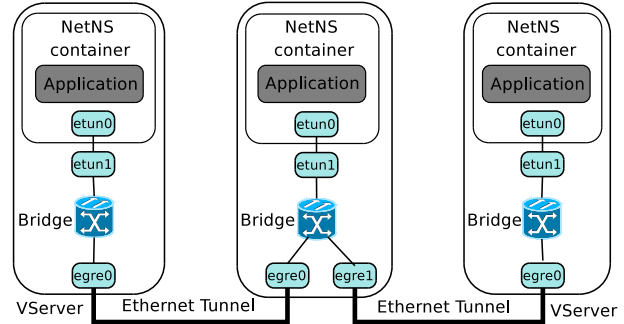
**Decision 3** *Terminate tunnels in the “root context”, outside of virtual host containers.*

Trellis’s virtual links must be isolated from links in other virtual networks (i.e., traffic on one virtual network cannot interfere with that on another), and they must be flexible (i.e., users must be able to specify many policies). To satisfy these goals, Trellis terminates virtual links in the root context, rather than in the virtual host contexts. Table 2 summarizes why we made this decision, with further detail below.

Terminating the tunnel in the root context, rather than inside the container, allows the infrastructure administrator to impose authoritative bandwidth restrictions on users. Applications running on a virtual host have full control over the environment in a container, including access to network bandwidth. To enforce isolation, Trellis must enforce capacity and scheduling policies *outside the container*. Trellis terminates tunnels in the root context; an intermediate queueing device between the tunnel interface and a virtual host’s virtual interface resides in the root context and shapes traffic using `tc`, the Linux traffic control module [22]. The virtual device inside the virtual host’s context is bridged with the tunnel endpoint. This arrangement allows them to apply traffic shaping policies and packet-filtering rules, and,

Criteria		In Container	In Root Context
Speed	Direct connection	Yes	No, needs bridging
Isolation	Enforceable bandwidth limits	No	Yes
Flexibility	Multi-point topologies	No	Yes
	User-defined shaping	Yes	Yes

**Table 2: Tradeoffs for terminating tunnel endpoints.**



**Figure 4: Bridging supports easy configuration of point-to-multipoint topologies.** A pair of `etun` interfaces are used to send ethernet frames from a network context into the root context. The Linux bridge module bridges the `etun` device with the EGRE tunnel interface.

ultimately to implement packet scheduling algorithms that provide service guarantees for each virtual interface. Users though can still apply their own traffic shaping policies on the virtual network interfaces inside their respective containers for their traffic.

Terminating the tunnel endpoints outside the network container also provides flexibility for configuring topologies. Specifically, this choice allows users to create point-to-multipoint topologies, as illustrated by Figure 4 and discussed in more detail in Section 4.3. It also allows containers to be connected directly when they are on the same host, instead of being forced to use EGRE tunnels.

### 4.3 Bridging: Bridge vs. Shortbridge

Our decision to terminate tunnels in the root context rather than in the host container itself creates the need to transport ethernet frames between the tunnel interface (in the root context) and the virtual interface (on a virtual host). One way to implement this is with *software bridging*, which is supported by Linux kernel or by some stand-alone virtualization solutions [39]. Like a traditional ethernet bridge, the software bridge performs a lookup on the destination MAC address and determines where to send the packet. Software bridge enables connecting interfaces together at Layer 2 with ethernet semantics.

We explore two options for bridging EGRE tunnels to virtual interfaces: (1) the standard Linux *bridge* module [3]; and (2) *shortbridge*, a custom, high-performance device that we implemented specifically for bridging a single virtual interface directly to its corresponding tunnel interface. Each option offers different benefits: the bridge module offers additional *flexibility* in defining the network topology, while the shortbridges offers better *speed* (i.e., higher

Criteria		Bridge	Shortbridge
Flexibility	Multi-point topologies	Yes	No
Speed		No	Yes
Scalability		No	Yes

Table 3: Design tradeoffs for using bridge vs. shortbridge.

packet-forwarding rates). We use the standard Linux bridge in links that require point-to-multipoint connectivity; and *shortbridges* to maximize performance for interfaces that are connected to point-to-point links. Table 3 summarizes the tradeoffs, which we discuss in more detail in this section.

**Decision 4** When the virtual network topology requires point-to-multipoint links, connect tunnel interfaces with virtual interfaces using the Linux bridge.

**Flexibility** Some networks require bus-like, transparent multipoint topologies, where a set of interfaces can have the appearance of being on the same local area network or broadcast medium. In these situations, a broadcast or multicast packet sent from a single interface can reach all interfaces on the medium. The standard Linux bridge module makes configuring such a topology quite easy, because it can interconnect more than two interfaces. Figure 4 shows an example point-to-multipoint topology with the Linux bridge configuration. The three nodes perceive the underlying network as connected by a single switch.

In multipoint topology case, Trellis connects an EGRE tunnel to its corresponding virtual interface using (1) *etun*, a pair of devices that transports packets from a host container to the root context; and (2) the Linux bridge module, which emulates the behavior of a standard Layer 2 bridge in software and connects interfaces together inside the root context.

As shown in Figure 4, *etun* is instantiated as a pair of connected devices, of which one is located inside a user container (*etun0*) and the other, *etun1* is located in the root context. The pair of *etun* devices is necessary because the bridge lies in the root context and it must have an abstraction of an interface to bridge to. Frames sent via *etun0* arrive at *etun1*, and vice versa. The Linux bridge module connects the end of the virtual interface that resides in the root context, *etun1*, to the tunnel endpoint.

Unfortunately, as our experiments in Section 5 show, using the bridge module can degrade packet forwarding performance considerably, due to the overhead of copying the frame header, learning the MAC addresses, and performing the MAC address table lookup itself (*i.e.*, to determine which outgoing interface corresponds to the destination ethernet address). When network links are point-to-point, this lookup is unnecessary and can be short-circuited; this insight is the basis for the “shortbridge” optimization, which we describe next.

**Decision 5** When the virtual links are point-to-point, connect tunnel interfaces with virtual interfaces using the “shortbridge”.

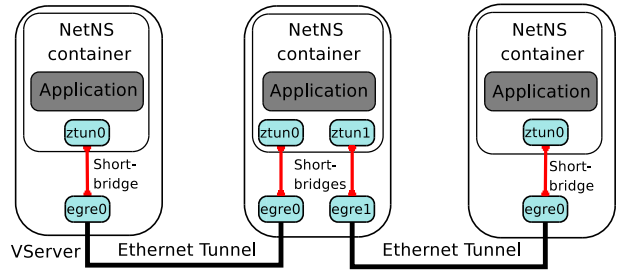


Figure 5: High speed forwarding using shortbridges: The shortbridge device is used to connect the *ztun* device located inside the container with the EGRE tunnel interface. Shortbridge avoids any lookups as performed by the bridge and hence improves forwarding speed.

**Speed** Forwarding packets between the virtual network interface and the tunnel interface must be fast, which implies that the bridge should determine as quickly as possible outgoing interface on which it could send the packet. A potential bottleneck for transporting traffic is thus the lookup at the bridge (*i.e.*, mapping the destination MAC address of the ethernet frame to an outgoing port). In the case of shortbridge, which connects only a pair of interfaces, this lookup is trivial and is thus very fast. The Linux bridge module, on the other hand, is slightly slower, since it performs several additional operations. For this reason, when packet-forwarding speed is paramount and the network topology need not support point-to-multipoint links, we opt to connect virtual interfaces to tunnel interfaces with the shortbridges.

We have implemented an optimized version of the bridge module called *shortbridge*. We have also implemented a new device, *ztun* which, unlike the *etun* device, is a *single* virtual interface inside the container that the shortbridge can connect directly to the tunnel interface without requiring a corresponding interface in the root context. The *ztun* interface is instantiated as a single interface inside a host container and connects directly to the shortbridge. Figure 5 shows a configuration using the shortbridge device; a single shortbridge device connects one virtual interface (*i.e.*, *ztun* device) to one tunnel interface (*i.e.*, *egre* device).

We achieve performance gain with shortbridge, because no bridge table lookup is required: traffic can simply be forwarded from the single *egre* device to the single *ztun* device, and vice versa. Second, the configuration avoids an extra header copy operation by reusing the packet data structure for the two devices that are connected to the shortbridge. Third, this pair of devices is very restricted: the *ztun* device always connects to a tunnel endpoint; thus, shortbridge maintains a pre-defined device-naming scheme which allows each *ztun/etun* pair to have a static mapping, avoiding potentially slow lookups.

## 5. Evaluation

This section evaluates whether Trellis satisfies our three design goals: *forwarding performance*, *scalability*, and *isolation*. We focus in particular on Trellis’s packet-forwarding performance compared to other possible environments for building virtual networks, including Xen, OpenVZ, and for-

warding in user space. Our experiments show that Trellis can provide packet-forwarding performance that is about 2/3 of kernel-level packet forwarding rates, which is nearly a ten-fold improvement over previous systems for building virtual networks [10]. The rest of this section describes the experimental setup and the detailed results of our performance evaluation.

## 5.1 Setup

**Test Nodes** We evaluated the performance of Trellis and other approaches using the Emulab [42] facility. The Emulab nodes are connected through a switched network. All connections offer stable 1 Gbps speeds and negligible, LAN-level delays. The Emulab nodes used were Dell Poweredge 2850 servers with 3.0 GHz 64-bit Intel Xeon processor with 1MB L2 cache, 800 MHz FSB, 2GB 400MHz DDR2 RAM and two Gigabit ethernet interfaces. We used a customized 2.6.20 Linux kernel patched with Linux VServer and NetNS support and used the Redhat Linux distribution. The kernel also includes our custom kernel patches to provide support for EGRE and shortbridge.

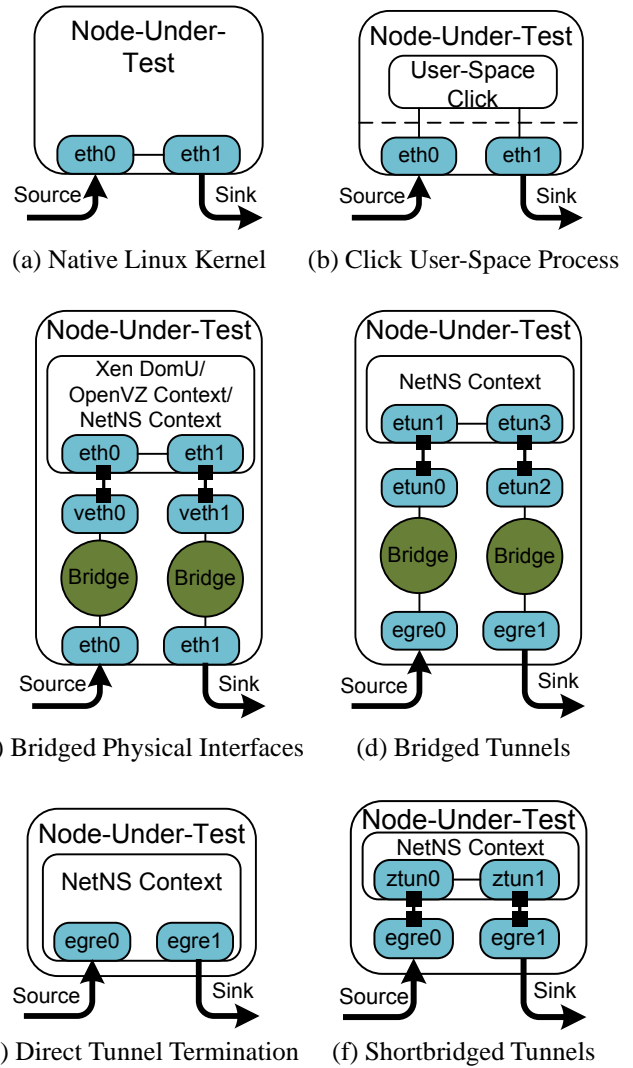
**Traffic Generation** Popular network performance tools such as *iperf* or *netperf* are not sufficient for our needs, because these tools generate packets from user space which can hardly exceed more than 80,000 packets per second (pps). Instead, we generated traffic using *pktgen* [28], a kernel module that generates packets at a very high rate. It bypasses the networking stack to directly interact with the NIC and includes other techniques to optimize memory allocation for packet generation. Although it is fast, *pktgen* offers poor rate control at very high speeds. We re-ran all experiments several times to confirm that the results were not affected by *pktgen*'s rate control problems.

The Linux kernel packet-forwarding rate follows the standard system-load curve: under increasing load system performance improves; after a certain threshold it becomes less effective due to overhead of processing multiple service requests. To determine the peak performance, we gradually varied load from high to low and noted the peak throughput.

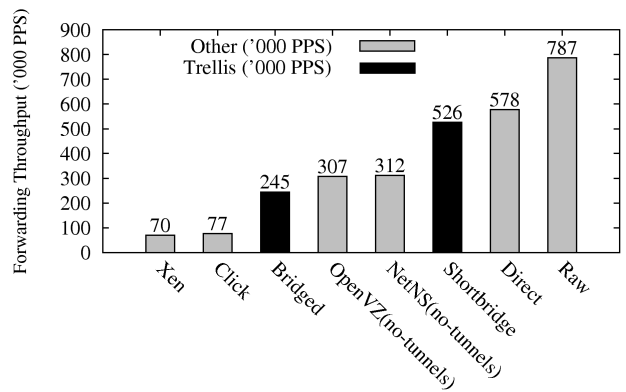
**Virtualization Environments** In addition to the standard Trellis setup, we evaluated the performance of network virtualization by performing experiments in both a full virtualization environment (*i.e.*, Xen), two container-based virtualization environments (*i.e.*, Trellis and OpenVZ), and Click tunnels in user space. Figure 6 summarizes the experimental setup for each of these experiments; we discuss these in more detail in Section 5.2.

## 5.2 Forwarding Performance

In this section, we present the forwarding performance (in terms of pps) for various virtualization technologies. We performed packet-forwarding experiments for all of the environments shown in Figure 6 (including Xen, OpenVZ, and NetNS in the case of Figure 6(d) and compared each of these to the baseline forwarding performance of the native Linux 2.6.20 kernel. We established the topology simply by installing static routing table entries in the kernel routing table,



**Figure 6: Experiment Setup.** Each setup has a source, a sink and a node-under-test. The traffic from the source arrives on the physical interfaces in setups (a),(b) and (c), while in setups (d), (e) and (f) the source traffic goes through the tunnel interfaces.



**Figure 7: Peak forwarding performance (in pps) with 64-byte packets.**

as shown in Figure 6(a). We evaluated scaling and isolation performance only for Trellis.

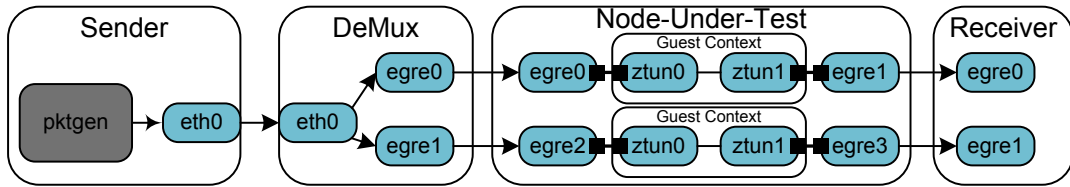


Figure 8: Setup For Scalability Evaluation. This setup is also used for the isolation experiment using multiple containers.

The goals for our experiments are two-fold. First, we study the performance of various virtualization techniques and their suitability for building virtual networks that can forward packets at high rates. We quantify the performance overhead associated with building virtual networks with user-space packet processing or full virtualization. Second, we evaluate the packet-forwarding performance for various network designs that use container-based virtualization.

### 5.2.1 Comparison of virtualization approaches

**Click in user space** To evaluate the baseline performance of forwarding packets in user space, we forwarded traffic through a Click user-space process, as in the original PL-VINI environment [10], as shown in the Figure 6(b). Click offers flexible primitives for packet manipulation and forwarding. It can run as a user-space process or a kernel-space module. A kernel-space module cannot properly allocate resources, because it exposes the whole kernel memory space to any Click element. Our earlier PL-VINI [10] implementation successfully used a user-space Click process, but this setup did not achieve adequate forwarding speed.

We used a simple, lightweight Click `Socket()` element to forward UDP packets. Figure 7 shows that the peak packet-forwarding rate for 64-byte packets was approximately 80,000 pps. PL-VINI sustained even worse performance because it used a large set of Click elements with complex interactions between them.

**Full Virtualization: Xen** We measured the forwarding performance of Xen by running Xen 3.0.2 on the node under test with one guest domain. We bridged the virtual interfaces in DomU (the user domain) to the physical interfaces in the privileged domain, Dom0, using the Linux bridge module, as shown in Figure 6(c). We swapped in Xen from the Emulab system image repository. The Emulab images are specifically compiled for Emulab nodes. Unfortunately, we found Xen 3.0.2 unstable under high packet load, which is consistent with observations in other studies [25, 27]. Packet rates of more than 70,000 pps resulted in unstable behavior.<sup>1</sup> Recent activity in the Xen community suggests that newer versions of Xen might have a more stable network stack that offers better network performance [25]; we intend to evaluate these alternatives in the future.

**Container-Based Virtualization: OpenVZ and Trellis** We evaluated OpenVZ to compare Trellis’s performance

<sup>1</sup>After about 15 seconds of such load, the DomU virtual interfaces stopped responding. Increasing the traffic load further, to more than 500,000 pps, caused the hypervisor to crash. We repeated the experiment with the same setup and similar hardware on our own nodes and found similar behavior.

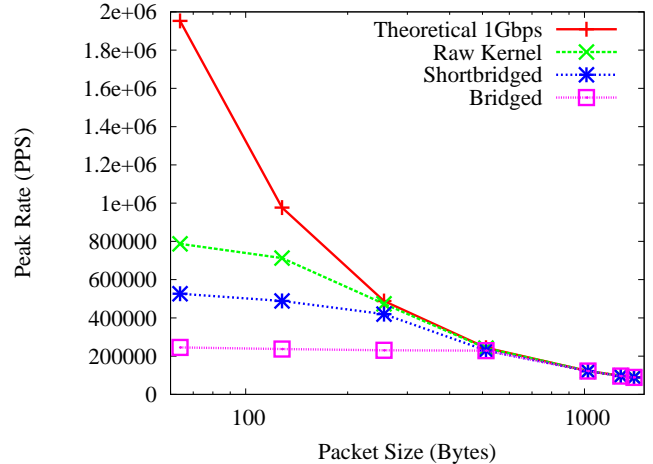


Figure 9: Peak forwarding rate (in pps) for different packet sizes.

with another container-based virtualization system. OpenVZ does not provide EGRE or shortbridge features; thus, we connected the nodes directly, without tunnels and used a regular bridge module to connect the physical interfaces to the virtual interfaces. Figure 6(c) shows our configuration for the OpenVZ setup and for a Trellis setup with no EGRE tunnels and a regular bridge module (*i.e.*, NetNS+VServer); this setup is analogous to our setup for the forwarding experiment with Xen.

Figure 7 shows that the performance of OpenVZ is comparable to that of Trellis when plain ethernet interfaces and bridging are used; with this configuration, both systems achieve peak packet-forwarding rates of approximately 300,000 pps. This result is not surprising, because both OpenVZ and Trellis have similar implementations for the network stack containers. This result suggests that Trellis could be implemented with OpenVZ, as opposed to VServers+NetNS, and achieve similar forwarding rates.

### 5.2.2 Optimizing container-based virtualization

We evaluate the effects of various design decisions within the context of container-based virtualization: In addition to the five environments above, we evaluated various optimizations and implementation alternatives within the context of Trellis. Specifically, we examined the effects of (1) where the tunnel terminates and (2) using bridge vs. shortbridge on both packet-forwarding performance and isolation.

**Overhead of terminating tunnels outside of container** Directly terminating EGRE tunnels inside the container context *inside the container context*. This approach provides

little control over the network resources that the container uses (*i.e.*, it is not possible to schedule or rate-limit traffic on the virtual links), but it offers better performance by saving a bridge table lookup. To quantify the overhead of terminating tunnels outside of containers, we perform a packet forwarding experiment with the configuration shown in Figure 6(e).

Figure 7 summarizes these results. Directly terminating the tunnels within the container (Figure 6(e)) achieves a packet-forwarding rate of 580,000 pps (73% of native forwarding performance); as mentioned, however, this mechanism does not provide the capability to shape or otherwise control outside the container. This performance gap directly reflects the overhead of network-stack containers and EGRE tunneling.

**Bridge vs. Shortbridge** To evaluate the performance improvement of the shortbridge configuration over the standard Linux bridge module, we evaluate packet-forwarding performance in the following two setups:

- **Bridge.** Figure 6(d) shows the setup of bridged experiment for Trellis. A similar setup is used for evaluating forwarding performance in Xen and OpenVZ where a bridge is used. However, in Xen and OpenVZ, the bridge joins virtual environment (or virtual machine in the case of Xen) with the physical interfaces on the node, but in Trellis the bridge connects the virtual environment to EGRE tunnels.
- **Shortbridge.** We replace the Linux bridge module with our custom high-performance forwarding module *shortbridge* to connect virtual devices with their corresponding physical devices, as shown in Figure 6(f). We perform this experiment to determine the performance improvement over the regular bridging setup.

The shortbridge configuration achieves a forwarding rate of 525,000 pps (about 67% of native forwarding performance). The performance gain over the bridge configuration results from avoiding both copying the ethernet frame an extra time, as well as performing bridge table lookup for each ethernet frame. The bridged setup can forward packets at around 250,000 pps.

### 5.2.3 Effects of packet size on forwarding rate

Figure 9 shows how the packet-forwarding rate varies with packet size, for the bridge and shortbridge configurations, with respect to the theoretical capacity of the link and the raw kernel forwarding performance. The nearly flat lines in the bridged and shortbridged configurations indicate that Trellis packet-forwarding performance does not change much as the packet size changes. This result is expected, because all packet processing happens in-kernel and most operations are performed only on the packet header. For larger packets, the rate is limited by the 1 Gbps link. Trellis’s packet-forwarding performance with the shortbridge configuration approaches the performance of native forwarding for 256-byte packets; for 512-byte and larger packets, both the bridge and shortbridge configurations saturate the outgoing 1 Gbps link.

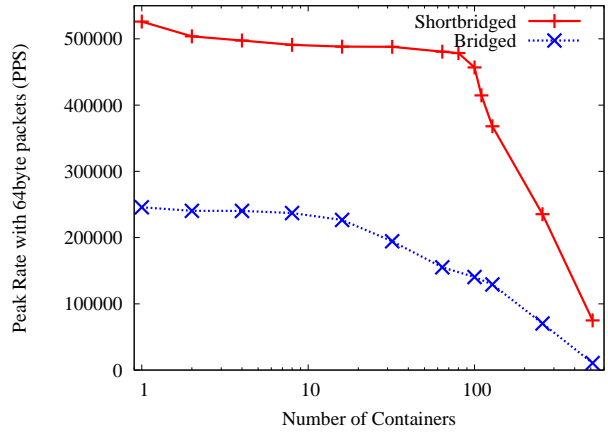


Figure 10: Scalability test. Peak forwarding rate (in packets per second) for 64-byte packets for different number of concurrent containers.

## 5.3 Scalability

We evaluate scalability of the bridge and shortbridge configurations in Trellis by increasing the number of containers on the single physical node under test and measuring the corresponding throughput of the resulting flows. Figure 8 shows the configuration that we used to test the scalability of Trellis’s container mechanism for both the bridge and shortbridge configurations. As with the packet-forwarding rate experiments, we use the bridged and shortbridged setups as shown in Figure 6(d) and Figure 6(f), respectively.

Our tests show that Trellis can support at least 64 concurrent virtual networks without a noticeable degradation in performance. As can be seen from Figure 10, in the case of the shortbridged configuration, the forwarding performance decreases from 525,000 pps with one container to 70,000 pps with 512 containers. In the bridged configuration, the drop is more dramatic because each container uses two bridges. The rate starts at 250,000 pps with one context and ends with 10,000 pps combined throughput for 512 containers. Profiling the bridged configuration showed that several factors introduced overhead, including checking the consistency of the ethernet header, copying the packet header, and performing a table lookup and update. Furthermore, as the number of containers increases, the overhead for forwarding, encapsulation and decapsulation consumes most of the CPU.

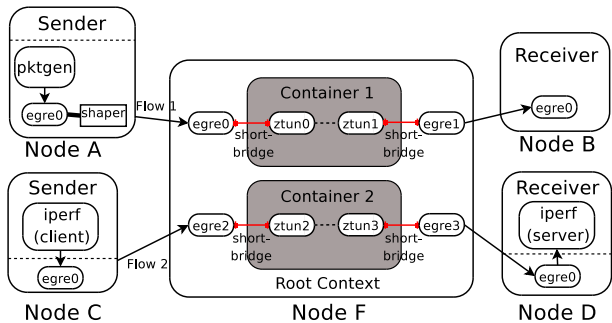
Another notable feature is the sharp drop in the forwarding rate with shortbridge when the number of containers is between 64 and 80. There are many possible explanations for this degradation, including shared data structures, limited hardware caches, etc. We are still investigating the root cause of this behavior.

## 5.4 Isolation

To understand how Trellis would perform in scenarios with many virtual networks operating in parallel, we evaluate how Trellis provides isolation with multiple virtual networks running concurrently. Our results show that Trellis can prevent traffic on one virtual network from interfering with traffic from other virtual networks that are using the

Flow 1 (kpps)	No other load			Disk-bound			CPU-bound		
	CPU (%)	Loss (%)	Jitter (ms)	CPU (%)	Loss (%)	Jitter (ms)	CPU (%)	Loss (%)	Jitter (ms)
0	20	0	0.030	31	0	0.030	18 (82)	0	0.030
500	92	0	0.049	100	0	0.080	97 (3)	0	0.102
800	100	25	0.063	100	26	0.120	100 (0)	32	0.110

**Table 4:** This table summarizes the results from the isolation experiment using setup as shown in Figure 11. We performed three different experiments: when the forwarding node had no other load, a disk bound process and a CPU bound process running. The rows show the results for each experiment for three different rates of Flow 1. For the CPU-bound experiment, the numbers for CPU usage in parenthesis shows the CPU utilization by the CPU-bound process.



**Figure 11:** Setup for the isolation experiment.

same physical network. We evaluated isolation of experiments under various load conditions of the forwarding node.

Figure 11 shows the topology that we used for the first part of the experiment. Traffic flows from node *A* to node *B* (Flow 1) and from node *C* to node *D* (Flow 2). All traffic uses node *F* for forwarding. Node *F* has two containers: one container serves traffic for Flow 1, and the other container serves traffic on Flow 2. The setups uses the shortbridge to connect virtual interfaces with physical interfaces and EGRE tunnels to connect nodes. We used the Linux `pktgen` module to generate packets for Flow 1 at a rate that is close to the forwarding capacity of node *F*. We used the `iperf` utility to generate Flow 2. We varied the traffic rate of Flow 1 using the `traffic control` [22] utility in Linux to shape the traffic and measured the resulting jitter and loss rate on as seen by the receiver at Flow 2. We used a hierarchical token bucket (HTB) filter to control the offered traffic load on Flow 1.

Table 4 summarizes the results of this experiment. When the CPU utilization at the forwarding node *F* was below 100%, Flow 2 experienced no loss and only negligible jitter. When Flow 1 sends 800 kpps (800 pps), CPU utilization at *F* becomes 100%, Flow 2 experiences loss, but jitter is still negligible for packets that *F* did not drop. To measure the possible impact of other interference, we ran two additional experiments: a disk-bound I/O process (*i.e.*, generating hard and soft interrupts) on the forwarding node *F*, and a CPU-bound process on *F*. In both cases, the loss and jitter exhibited the same behavior as in the original experiment: as long as the amount of CPU load due to forwarding packets remained below 100%, packet loss and jitter were negligible. We were not surprised that a CPU-bound process had no impact (since in-kernel packet forwarding preempts a running user-space process) but we expected to see more

interference from disk-bound I/O. In our ongoing work, we are studying the isolation properties of Trellis in more detail and in more complex deployment scenarios.

We also evaluated the effects increasing the number of containers carrying traffic on the jitter induced by the forwarding node *F* by establishing 32 concurrent containers, introducing background traffic on the other 31 containers, and measuring the jitter on one remaining flow at the receiver. For this experiment, we use the same setup as shown in Figure 8. We did not observe any significant jitter; the results were consistent with the ones described in Table 4. Due to our inability to obtain a very large cluster of physical nodes, all traffic flows—both the flow on which the resulting jitter was measured was sent from the same nodes and the background traffic—were sent from the same physical machine, which may have affected the jitter seen at the receiver in addition to any contributions to jitter at the forwarding node. We intend to investigate this result in more detail.

## 6. Ongoing Work

This section describes our ongoing work on Trellis. We describe ongoing work in three areas: supporting high-performance custom data planes, managing and allocating physical resources, and implementing link scheduling disciplines to achieve better isolation.

### 6.1 Supporting Custom Data Planes

The current Trellis implementation uses container-based virtualization. Thus, building a virtual network with a custom data plane (either in Trellis or in any virtual network environment) requires forwarding packets either through user space or, in the case of Xen, a driver domain; these approaches are significantly slower than forwarding packets in the kernel. Many virtual network deployments and experiments will not require custom data-plane operations (*e.g.*, non-IP forwarding), but we believe that ultimately some of them will require a custom data plane.

We are investigating mechanisms that would allow virtual hosts to define custom data planes and still achieving fast performance. One option is to use Linux kernel-based virtual machines, which exploit hardware support for virtualization to achieve performance that is close to that of the native operating system [2]. KVM has a special guest mode that, through a user-space emulator, interacts with a device driver for managing the virtualization hardware. I/O in KVM avoids costly context switches and interrupt processing, which could make it possible to implement a cus-

tom data plane by running custom packet-processing software (*e.g.*, Click-based applications) as a KVM guest. It remains to be seen whether this model can provide both flexibility and fast packet forwarding.

Another alternative for supporting custom data planes would be to sub-divide the kernel memory itself, dedicating in-kernel memory to explicit processes in user space (*i.e.*, dedicating memory to each container). This model is analogous to the approach that Supercharged PlanetLab (SPP) has taken to virtualize forwarding hardware [37]. This approach would create a static mapping between each virtual host and kernel data structures; an experiment’s special data structures could be defined by the user using a Click-like specification, checked at compile-time, and installed as a kernel module.

## 6.2 Resource Management and Allocation

This paper has described the mechanisms by which a virtual network can be created and hosted on shared, commodity hardware, but it has not tackled the allocation and management of the physical resources. This management requires two components: (1) a system for maintaining information about the “inventory” of the physical network and what resources have been allocated; and (2) algorithms for embedding a virtual network topology on to the physical infrastructure.

To solve the problem of maintaining information about available physical resources, we envision a software distribution for Trellis that is analogous to PlanetLab’s MyPLC [4]. The MyPLC distribution packages the same software base that operates the public PlanetLab facility. Like PlanetLab, MyPLC allocates VServers to collections of physical nodes and maintains information about how resources have been allocated on each node, as well as statistics about usage and load on each node. Trellis’s management system would be slightly more complicated because it must maintain information about allocations on network links (where actual usage can be quite variable). Ultimately, we plan to package Trellis, including software for resource management, as a standalone distribution that anyone can use to operate and own a virtual network hosting platform.

The network manager must determine an allocation of resources that satisfies the requests of many virtual networks running in parallel. This allocation boils down to an embedding problem: given an underlying physical topology (physical nodes, and links), and potentially many requests for virtual networks to be hosted on the infrastructure, determine the best way to embed the virtual topologies onto the underlying physical topology. The embedding problem is challenging because it involves allocation of many resources, including CPU, link bandwidth, and memory. Additionally, to maximize utilization, the allocation mechanism might try to “overbook” links by taking advantage of statistical multiplexing across virtual networks. A few recent attempts to tackle embedding a virtual network topology on a physical topology [30, 44] suggest possible starting points for the network embedding problem in Trellis.

## 6.3 Resource Scheduling and Isolation

Although our experiments in Section 5.4 suggest that Trellis already provides good resource isolation, we believe that more complex usage scenarios and applications, as well as larger virtual networks, might still trigger cases where individual virtual networks see behavior that differs from that which it would see if the network were running on dedicated hardware.

One example that deserves further study is the scheduling of virtual links. Trellis’s traffic shapers on individual interfaces can ensure that no virtual interface exceeds its allocated capacity; unfortunately, this traffic shaping does *not* guarantee that each virtual link will see packet service rates that equate to those it would see in the case that it had a dedicated link. For example, some scheduling disciplines might cause traffic in some virtual networks to experience temporary starvation, introducing jitter or packet loss. Although the container-based virtualization community has made some progress in ensuring that each container sees some minimum level of performance (*e.g.*, OpenVZ’s Bean-counters [14]), more work is needed to determine appropriate service disciplines for scheduling the outgoing physical interface to different tunnel interfaces. Previous work on the performance guarantees offered by various service disciplines [43], which were originally designed for multi-hop end-to-end paths, might apply for scheduling service on a single virtual link.

## 7. Conclusion

This paper has presented Trellis, a platform for scalably hosting virtual networks on commodity hardware. Trellis allows each virtual network to define its own topology, routing protocols, and forwarding tables, thus lowering the barrier for enterprises and service providers to define custom networks that are tailored to specific applications or users. Trellis uses network virtualization to allow multiple virtual networks to share the same physical infrastructure. The platform integrates host and network stack virtualization with tunneling technologies and our own components, EGRE tunnels and shortbridge to create a coherent framework for building fast, flexible virtual networks.

While many of these technologies have existed for some time, we believe our combination of these components provides not only a useful system but also important lessons for combining various host and link virtualization technologies to build virtual networks. We have implemented Trellis using Linux-VServer, NetNS, and EGRE tunnels and compared the speed, flexibility, and isolation of our system with existing techniques for building virtual networks. While additional work remains to improve Trellis’s resource management, allocation and scheduling, we believe our design to be a promising step for building virtual networks from commodity hardware. Ultimately, we hope that Trellis can be for virtual routers and networks what software routers have done for routing: a scalable, fast, low-cost alternative for deploying virtual networks in practice.

## REFERENCES

- [1] Kernel-based Virtual Machines. <http://kvm.gumranet.com/>.
- [2] KVM: Kernel-based Virtualization Driver. [http://www.gumranet.com/wp/kvm\\_wp.pdf](http://www.gumranet.com/wp/kvm_wp.pdf).
- [3] Linux BRIDGE-STP-HOWTO. <http://www.faqs.org/docs/Linux-HOWTO/BRIDGE-STP-HOWTO.html>.
- [4] MyPLC. <http://www.planet-lab.org/doc/myplc>.
- [5] Quagga software router. <http://www.quagga.net/>, 2006.
- [6] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. Symposium on Operating Systems Principles*, pages 131–145, October 2001.
- [7] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. Symposium on Operating Systems Design and Implementation*, pages 45–58, 1999.
- [8] M. Bauer. Paranoid penguin: Linux vpn technologies. *Linux J.*, 2005(130):13, 2005.
- [9] A. Bavier, M. Bowman, D. Culler, B. Chun, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. Networked Systems Design and Implementation*, March 2004.
- [10] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. ACM SIGCOMM*, Pisa, Italy, August 2006.
- [11] E. Biederman. NetNS. <https://lists.linux-foundation.org/pipermail/containers/2007-September/007097.html>.
- [12] Cisco Systems: Inter-AS/Carrier Supporting Carrier. [http://www.cisco.com/en/US/products/ps6650/products\\_ios\\_protocol\\_option\\_home.html](http://www.cisco.com/en/US/products/ps6650/products_ios_protocol_option_home.html).
- [13] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. Symposium on Operating Systems Principles*, October 2003.
- [14] P. Emelianov, D. Lunev, and K. Korotaev. Resource Management: Beancounters. In *Proc. Linux Symposium, Ottawa, Canada*, June 2007.
- [15] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. *Generic Routing Encapsulation (GRE)*. Internet Engineering Task Force, March 2000. RFC 2784.
- [16] Q. W. Group. IEEE Standard for Local and Metropolitan area networks: Virtual Bridged Local Area Networks". IEEE Std 802.1Q–2005, May 2006.
- [17] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *Proc. Symposium on Operating Systems Design and Implementation*, December 2004.
- [18] M. Handley, O. Hudson, and E. Kohler. XORP: An Open Platform for Network Research. In *Proc. SIGCOMM Workshop on Hot Topics in Networking*, pages 53–57, October 2002.
- [19] M. Handley, O. Hudson, and E. Kohler. XORP: An open platform for network research. In *Proc. SIGCOMM Workshop on Hot Topics in Networking*, October 2002.
- [20] P. Kamp and R. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd Intl. SANE Conference*, 2000.
- [21] S. Kent, C. Lynn, J. Mikkelsen, and K. Seo. Secure border gateway protocol (S-BGP) - real world performance and deployment issues. In *Proc. Network and Distributed Systems Security, Internet Society*, 2000.
- [22] Linux Advanced Routing and Traffic Control. <http://lartc.org/>.
- [23] Linux VServers Project. <http://linux-vserver.org/>.
- [24] Juniper Networks: Intelligent Logical Router Service. [http://www.juniper.net/solutions/literature/white\\_papers/200097.pdf](http://www.juniper.net/solutions/literature/white_papers/200097.pdf).
- [25] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing Network Virtualization in Xen. In *Proc. USENIX Annual Technical Conference*, pages 15–28, 2006.
- [26] OpenVZ: Server Virtualization Open Source Project. <http://www.openvz.org>.
- [27] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. Technical Report HPL-2007-59, HP Labs, April 2007.
- [28] pktgen: Linux packet generator tool. <http://linux-net.osdl.org/index.php/Pktgen>.
- [29] S. Ratnasamy et al. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, San Diego, CA, August 2001.
- [30] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM Computer Communication Review*, 33(2):65–81, 2003.
- [31] E. Rosen and Y. Rekhter. *BGP/MPLS VPNs*. Internet Engineering Task Force, March 1999. RFC 2547.
- [32] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.
- [33] S. Soltesz, H. Pözl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. EuroSys*, pages 275–287, 2007.
- [34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, San Diego, CA, August 2001.
- [35] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstations Hosted Virtual Machine Monitor. In *Proc. USENIX Annual Technical Conference*, pages 1–14, 2001.
- [36] A. Tucker and D. Comay. Solaris Zones: Operating System Support for Server Consolidation. *3rd Virtual Machine Research and Technology Symposium Works-in-Progress*.
- [37] J. Turner et al. Supercharging PlanetLab: A high performance, multi-application, overlay network platform. In *Proc. ACM SIGCOMM*, pages 85–96, Kyoto, Japan, August 2007.
- [38] SWSOft: Virtuozzo Server Virtualization. <http://www.swsoft.com/products/virtuozzo>.
- [39] VMware, Inc. VMware virtual machine technology". <http://www.vmware.com>.
- [40] VTun - Virtual Tunnels. <http://vtun.sourceforge.net>.
- [41] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. Symposium on Operating Systems Design and Implementation*, December 2002.
- [42] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. Symposium on Operating Systems Design and Implementation*, pages 255–270, December 2002.
- [43] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. In *Proceedings of the IEEE*, volume 83, pages 1374–1396, 1995.
- [44] Y. Zhu and M. Ammar. Algorithms for Assigning Substrate Network Resources to Virtual Network Components. In *Proc. IEEE INFOCOM*, 2006.