Trellis: A Platform for Building Flexible, Fast Virtual Networks on Commodity Hardware

Sapan Bhatia*, Murtaza Motiwala†, Wolfgang Mühlbauer‡, Yogesh Mundada†,
Vytautas Valancius†, Andy Bavier*, Nick Feamster†, Larry Peterson*, and Jennifer Rexford*
* Princeton University † Georgia Tech ‡ T-Labs/TU Berlin

ABSTRACT

We describe Trellis, a platform for hosting virtual networks on shared commodity hardware. Trellis allows each virtual network to define its own topology, control protocols, and forwarding tables, 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 and evaluate its packet-forwarding rates relative to other virtualization technologies and native kernel forwarding performance.

1. Introduction

Network researchers need a platform for testing new network architectures, protocols, and services. Although existing infrastructures like PL-VINI [4] can run multiple networking experiments in parallel, forwarding packets in user space significantly limits scalability. In addition to a realistic, controlled experimental setting, network researchers need a testbed that provides the following properties:

- *Speed.* The platform should forward packets at high rates. For example, if the platform forwards packets in software, the packet forwarding rates should approach that of "native" kernel packet forwarding rates.
- *Flexibility.* The platform should allow experimenters to modify routing protocols, congestion control parameters, forwarding tables and algorithms, and, if possible, the format of the packets themselves.
- *Isolation.* The platform should allow multiple experiments to run simultaneously over a single physical infrastructure without interfering with each other's namespaces or resource allocations.

This paper presents the design, implementation, and evaluation of *Trellis*, a platform that aims to find a "sweet spot" for achieving these three design goals, given that it is difficult to achieve all three simultaneously.[1] The main question we address in our evaluation is the extent to which we can provide experimenters both flexibility and speed, without compromising forwarding performance. Many existing "building blocks" can provide functionality for implementing the two key components of a virtual network (*i.e.*, virtual nodes and virtual links). Our main challenge is *synthesizing* existing mechanisms for implementing virtual nodes and virtual links in a manner that achieves the design goals above.

[1] More details are in the corresponding technical report [5].

Trellis is similar to the enhanced Emulab testbed features for virtualizing nodes; we are collaborating with the Emulab developers to integrate Trellis with the Emulab testbed. Emulab has focused mostly on resource allocation [9]; in contrast, we focus on the mechanisms for implementing the virtual network components.

To implement *virtual nodes*, two options are virtual machines (*e.g.*, Xen) and "container-based" operating systems (*e.g.*, VServer, OpenVZ). Container-based operating systems provide isolation of filesystem and the network stack without having to run an additional (potentially heavy-weight) instance of a virtual machine for each experiment. Trellis's container-based OS approach provides experimenters *flexibility* by allowing them to customize some aspects of the IP network stack (*e.g.*, congestion control) by giving each virtual network its own network namespace. In the current implementation, processing "custom" non-IP packets requires sending packets to user space, as in PL-VINI. In this paper, we evaluate the *speed* of this approach; in future work, we plan to study isolation, as others have recently done for full virtualization technologies [7].

Tunneling is a natural mechanism for implementing *virtual links*; unfortunately, existing tunneling mechanisms do not provide the appearance of a direct layer-two link, which some experiments might need. To solve this problem, we implement an *Ethernet GRE* (EGRE) tunneling mechanism that gives a virtual interface the appearance of a direct Ethernet link to other virtual nodes in the topology, even if that virtual link is built on top of an IP path.

Finally, Trellis must connect virtual nodes to virtual links; existing mechanisms, such as the bridge in the Linux kernel, allows virtual interfaces within each virtual node to be connected to the appropriate tunnels. To improve forwarding performance, we propose an optimization called *shortbridge*, which improves forwarding performance over the standard Linux bridge by avoiding unnecessary look-ups on MAC addresses and copying of frame headers.

The rest of the paper is organized as follows. Sections 2 and 3 describe the Trellis design and implementation, respectively. Section 4 compares Trellis's forwarding performance relative to other approaches (*e.g.*, virtual machines), as well as to in-kernel packet forwarding performance. Section 5 concludes and describes our ongoing work.

2. Trellis Requirements and Design

A *virtual network* comprises two components: **virtual hosts**, which run software and forward packets; and **virtual**

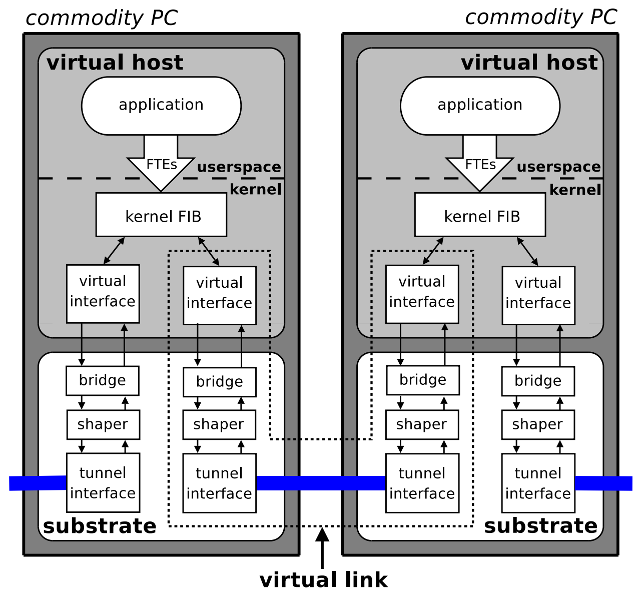


Figure 1: Overview of Trellis design.

links, which transport packets between virtual hosts. We describe the requirements for Trellis, as well as its high-level design. We then describe mechanisms for creating virtual hosts and links.

We identify four high-level design requirements for Trellis. First, 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 must run a *general-purpose operating system* inside the virtual hosts that can support existing routing software (e.g., XORP [8] and Quagga [3]) 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 to reduce system overhead and support higher packet-forwarding rates. An application running in user space 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.

Figure 1 illustrates a virtual network hosted on Trellis. The function of the virtual network is spread across three layers: user space inside the 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. Multiple virtual hosts can run on the same physical hardware (not shown in the figure). Physical network interfaces are also not shown because they are hidden behind the tunnel abstraction. 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.
- *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 challenge in building Trellis was to identify and combine individual virtual host and virtual link technologies that satisfied our design requirements, or implement new components in cases where existing ones did not meet the design requirements. The next section describes these design choices in the context of the Trellis implementation.

3. Trellis Implementation

Trellis synthesizes host and link virtualization technologies into a single, coherent system. In this section, we explain the implementation decisions we made when building Trellis to achieve our goals of speed, flexibility, and, where applicable, isolation.

3.1 Host Virtualization

Flexibility *Virtual hosts* must allow experimenters to implement both custom control-plane and data-plane functions, without compromising speed (*i.e.*, forwarding performance). Most types of host virtualization support control-plane customization; 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, this customization requires modifications to the guest OS. Container-based virtualization does not provide this flexibility because all virtual hosts share the same data structures in the kernel, but providing in-kernel data-plane customizability might ultimately be possible by partitioning kernel memory and data structures analogously to how similar systems have done this in hardware [10, 16].

In addition to providing fast forwarding and flexibility, Trellis should *scale*: it should support a large number of networks running simultaneously. Previous work, as well as our experiments in Section 4, 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.

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 [13, 15].

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

We combined two container-based approaches, Linux VServer [15] and NetNS [2], 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. NetNS virtualizes the entire Linux network stack, rather than simply providing each container with its own forwarding table. This enables Trellis to support experiments that want to configure, for example, TCP congestion-control parameters or IP packet manipulations; in addition, NetNS has recently been added to mainline Linux, making the use of NetNS especially appealing. Another possible choice for container-based host virtualization would have been OpenVZ, which has essentially the same functionality as our combination of VServer and NetNS; we evaluate both our approach and OpenVZ in Section 4.

3.2 Link Virtualization

Virtual links must 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 [6] for tunneling because it has a small, fixed encapsulation overhead and also uses a four-byte key to demultiplex packets to the right tunnel interface.

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

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). Forwarding non-IP packets would require running custom algorithms in user space, as in PL-VINI [4], or complex modifications to the kernel.

Speed 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. Trellis's EGRE-based tunneling approach is much faster than approaches that perform a lookup on the source, destination address pair. Other user-space tunneling technologies like `vtun` [17] impose considerable performance penalty compared to tunnels implemented as kernel modules.

Isolation 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.

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

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 [11]. 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, 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 discussed in more detail in Section 3.3. It also allows containers to be connected directly when they are on the same host, instead of being forced to use EGRE tunnels.

3.3 Bridging

Terminating tunnels in the root context rather than in the host container creates the need to transport ethernet frames between the tunnel interface (in the root context) and the virtual interface (on a virtual host). We explore two options for bridging EGRE tunnels to virtual interfaces: (1) the standard Linux *bridge* module [1]; 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 packet-forwarding rates). We use the standard Linux bridge for point-to-multipoint links; and *shortbridges* to maximize performance for interfaces that are connected to point-to-point links.

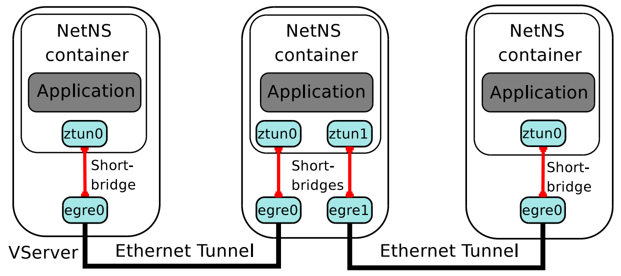


Figure 2: High speed forwarding using shortbridges: The shortbridge device is used to connect the $ztun$ device located inside the container with the EGRE tunnel interface. Shortbridge avoids any lookups as performed by the bridge and hence improves forwarding speed.

Decision 4 *For point-to-multipoint virtual links, connect tunnel interfaces with virtual interfaces using a 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 cases, 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. One `etun` device is located inside a user container (`etun0`) and the other, `etun1` is located in the root context; this configuration is necessary because the bridge lies outside of the container, yet it must have an abstraction of an interface to connect to for the corresponding device inside the container. The Linux bridge module connects the end of the virtual interface that resides in the root context to the appropriate tunnel endpoint.

Unfortunately, as our experiments in Section 4 show, using the bridge module slows packet forwarding due to additional operations: 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 described below.

Decision 5 *For point-to-point virtual links, connect tunnel interfaces with virtual interfaces using a "shortbridge".*

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 which outgoing interface should carry the traffic. 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).

For point-to-point links, 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 2 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).

Shortbridge achieves a performance speedup by avoiding a bridge table lookup: traffic can simply be forwarded from the single `egre` device to the single `ztun` device, and vice versa. 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. Additionally, shortbridge avoids an extra header copy operation by reusing the packet data structure for the two devices that are connected to the shortbridge.

4. Performance Evaluation

Ultimately, we aim to evaluate whether Trellis satisfies our design goals of *speed* and *isolation*. In this paper, we focus on speed, and specifically on Trellis's packet-forwarding performance compared to other environments, including Xen, OpenVZ, and forwarding 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 tenfold improvement over previous systems for building virtual networks [4].

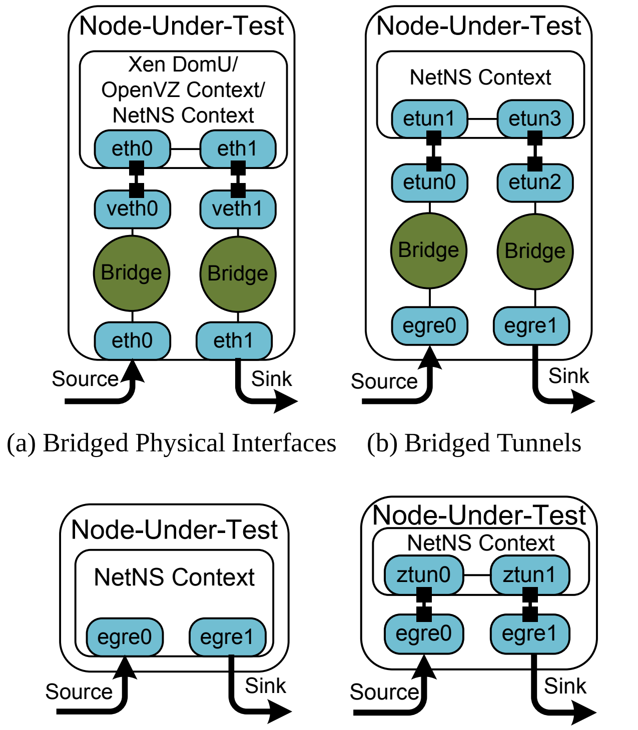
4.1 Experimental Setup

Test Nodes We evaluated the performance of Trellis and other approaches using the Emulab [18] facility. The Emulab nodes are connected through a switched network with stable 1 Gbps rates and negligible delays. The Emulab nodes 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 our custom kernel patches to provide support for EGRE and shortbridge.

Traffic Generation 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* [14], a kernel module that generates packets at a very high rate. We gradually varied load from high to low and noted the peak throughput.

4.2 Forwarding Performance

We evaluate the forwarding performance for various virtualization technologies. We performed packet-forwarding experiments for all of the environments shown in Figure 3 (including Xen, OpenVZ, and NetNS in the case of Figure 3(d) and compared each of these to the baseline forwarding performance of the native Linux kernel.



(a) Bridged Physical Interfaces (b) Bridged Tunnels

(c) Direct Tunnel Termination (d) Shortbridged Tunnels

Figure 3: 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 setup (a), while in setups (b), (c) and (d) the source traffic goes through the tunnel interfaces.

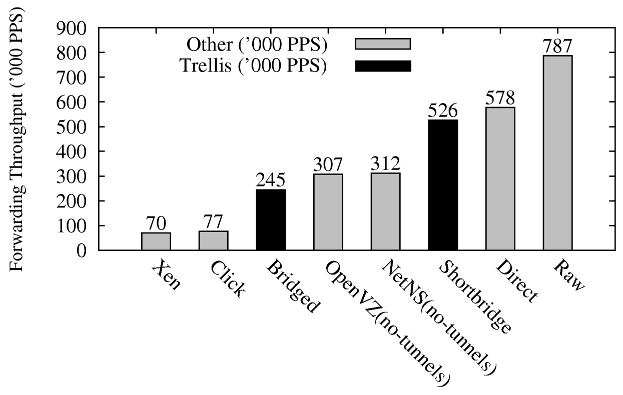


Figure 4: Peak forwarding performance (in pps) with 64-byte packets.

4.2.1 *Comparison of virtualization approaches*

User-Space Click 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 [4], as shown in the Figure 3(b). We used a simple, lightweight Click `Socket()` element to forward UDP packets. Figure 4 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.

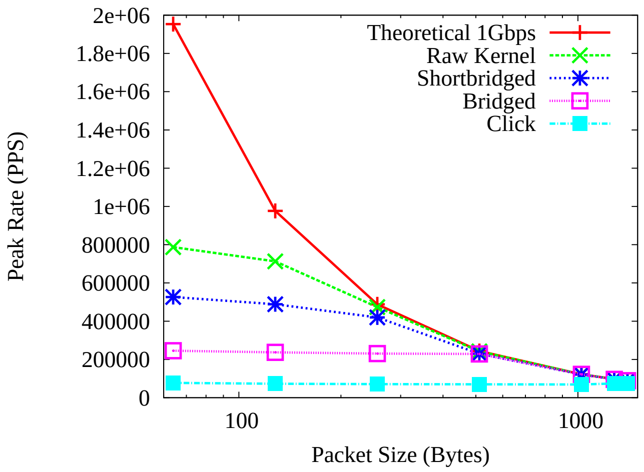


Figure 5: Peak forwarding rate (in pps) for different packet sizes.

Full Virtualization: Xen We measured the forwarding performance of Xen 3.0.2. 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 3(c). Unfortunately, Xen was unstable under packet rates of more than 70,000 packets per second, which is consistent with other studies [12, 13].[2] Recent activity in the Xen community suggests that newer versions might have a more stable network stack [12].

Container-Based Virtualization: OpenVZ and Trellis We evaluated OpenVZ to compare Trellis's performance 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 3(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 4 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.

4.2.2 *Optimizing container-based virtualization*

We evaluate the effects of various design decisions within the context of container-based virtualization: In addition to

[2]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.

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 provides the infrastructure administrator little control over the network resources that the container uses (*i.e.*, it is more challenging to schedule or rate-limit traffic on the virtual links). This approach also prevents the experimenter from directly changing parameters of the EGRE tunnel (e.g., the tunnel endpoints). However, terminating the tunnel inside the container could 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 3(e).

Figure 4 shows that directly terminating the tunnels within the container (Figure 3(e)) achieves a packet-forwarding rate of 580,000 pps (73% of native forwarding performance). 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 with two setups. First, Figure 3(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. In Xen and OpenVZ, the bridge joins the virtual environment with the physical interfaces on the node, but in Trellis the bridge connects the virtual environment to EGRE tunnels. Second, we replaced 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 3(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.

4.2.3 *Effects of packet size on forwarding rate*

Figure 5 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. For larger packets, the rate is limited by the 1 Gbps link. Trellis's packet-forwarding performance with shortbridge 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.

5. Conclusion

This paper has presented Trellis, a platform that 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 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.

REFERENCES

[1] Linux BRIDGE-STP-HOWTO. `http://www.faqs.org/docs/Linux-HOWTO/BRIDGE-STP-HOWTO.html`.

[2] Linux containers—network namespace. `http://lxc.sourceforge.net/network.php`.

[3] Quagga software router. `http://www.quagga.net/`, 2006.

[4] 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.

[5] S. Bhatia, M. Motiwala, W. Muhlbauer, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford. Hosting Virtual Networks on Commodity Hardware. Technical Report GT–CS–07–10, Department of Computer Science, Georgia Tech, 2008.

[6] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. *Generic Routing Encapsulation (GRE)*. Internet Engineering Task Force, March 2000. RFC 2784.

[7] A. Greenhalgh, M. Handley, L. Mathy, N. Egi, M. Hoerdt, and F. Huici. Fairness issues in software virtual routers. In *ACM SIGCOMM PRESTO Workshop*, Seattle, WA, aug 2008.

[8] 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.

[9] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In *Proc. USENIX*, Boston, MA, June 2008.

[10] E. Keller and E. Green. Virtualizing the Data Plane through Source Code Merging. In *ACM SIGCOMM PRESTO Workshop*, Seattle, WA, aug 2008.

[11] Linux Advanced Routing and Traffic Control. `http://lartc.org/`.

[12] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proc. USENIX Annual Technical Conference*, pages 15–28, 2006.

[13] 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.

[14] pktgen: Linux packet generator tool. `http://linux-net.osdl.org/index.php/Pktgen`.

[15] S. Soltesz, H. Pötzl, 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.

[16] 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.

[17] VTun - Virtual Tunnels. `http://vtun.sourceforge.net`.

[18] 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.