Building a Fast, Virtualized Data Plane with Programmable Hardware

Muhammad Bilal Anwer and Nick Feamster
School of Computer Science, Georgia Tech

ABSTRACT

Network virtualization allows many networks to share the same underlying physical topology; this technology has offered promise both for experimentation and for hosting multiple networks on a single shared physical infrastructure. Much attention has focused on virtualizing the network control plane, but, ultimately, a limiting factor in the deployment of these virtual networks is data-plane performance: Virtual networks must ultimately forward packets at rates that are comparable to native, hardware-based approaches. Aside from proprietary solutions from vendors, hardware support for virtualized data planes is limited. The advent of open, programmable network hardware promises flexibility, speed, and resource isolation, but, unfortunately, hardware does not naturally lend itself to virtualization. We leverage emerging trends in programmable hardware to design a flexible, hardware-based data plane for virtual networks. We present the design, implementation, and preliminary evaluation of this hardware-based data plane and show how the proposed design can support many virtual networks without compromising performance or isolation.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design C.2.6 [Computer-Communication Networks]: Internetworking

General Terms: Algorithms, Design, Experimentation, Performance

Keywords: network virtualization, NetFPGA

1. Introduction

Network virtualization enables many logical networks to operate on the same, shared physical infrastructure. Virtual networks comprise virtual nodes and virtual links. Creating virtual nodes typically involves augmenting the node with a virtual environment (*i.e.*, either a virtual machine like Xen or VMWare, or virtual container like OpenVZ). Creating virtual links involves creating tunnels between these virtual nodes (*e.g.*, with Ethernet-based GRE tunneling [5]). This technology potentially enables multiple service providers to share the cost of physical infrastructure. Major router vendors have begun to embrace router virtualization [7, 12, 14], and the research community has followed suit in building support for both virtual network infrastructures [3–5, 17] and services that could run on top of this infrastructure [10].

Virtual networks should offer good *performance*: The infrastructure should forward packets at rates that are comparable to a native hardware environment, especially as the number of users and virtual networks increases. The infrastructure should also provide strong *isolation*: Co-existing virtual networks should not interfere with one another. A logical approach for achieving both good performance and strong isolation is to implement the data plane in hardware. To date, however, most virtual networks provide only software support for packet forwarding; these approaches provide flexibility, ease of deployment, low cost, and fast deployment, but poor packet forwarding rates and little to no isolation guarantees.

This paper explores how programmable network hardware can help us build virtual networks that offer both flexibility and programmability while still achieving good performance and isolation. The advent of programmable network hardware (*e.g.*, NetFPGA [13, 18]), suggests that, indeed, it may be possible to have the best of both worlds. Of course, even programmable network hardware does not inherently lend itself to virtualization, since it is fundamentally difficult to virtualize gates and physical memory. This paper represents a first step towards tackling these challenges. Specifically, we explore how programmable network hardware—specifically NetFPGA—might be programmed to support fast packet forwarding for multiple virtual networks running on the same physical infrastructure. Although hardware-based forwarding promises fast packet forwarding rates, the hardware itself must be shared across many virtual nodes on the same machine. Doing so in a way that supports many virtual nodes on the same machine requires clever resource sharing. Our approach virtualizes the host using a host-based virtualized operating system (*e.g.*, OpenVZ [16], Xen [2]); we virtualize the data plane by multiplexing hardware resources.

One of the major challenges in designing a hardware-based platform for a virtualized data plane that achieves both high performance and isolation is that hardware resources are fixed and limited. The programmable hardware can support only a finite (and limited) amount of logic. To make the most efficient use of the available physical resources, we must design a platform that *shares* common functions that are common between virtual networks while still isolating aspects that are specific to each virtual network (*e.g.*, the for-

warding tables themselves). Thus, one of the main contributions of this paper is a design for hardware-based network virtualization that efficiently shares the limited hardware resources without compromising packet-forwarding performance or isolation.

We present a preliminary implementation and evaluation of a *hardware-based, fast, customizable virtualized data plane*. Our evaluation shows that our design provides the same level of forwarding performance as native hardware forwarding. Importantly for virtual networking, our design shares common hardware elements between multiple virtual routers on the same physical node, which achieves up to 75% savings in the overall amount of logic that is required to implement independent physical routers. Our preliminary design achieves this sharing without compromising isolation: the virtual router's packet drop behavior under congestion is identical to the behavior of a single physical router, but our implementation does not provide strong isolation guarantees to virtual router users; *i.e.*, if a user exceeds his bandwidth limit than other users are affected. Many issues remain to make our NetFPGA-based platform fully functional—in particular, we must augment the design to support simultaneous data-plane forwarding and control-plane updates, and design scheduling algorithms to ensure fairness and isolation between virtual networks hosted on the same hardware. This paper represents an initial proof-of-concept for implementing hardware-accelerated packet forwarding for virtual networks.

The rest of this paper is organized as follows. Section 2 provides background and related work on both programmable hardware and the NetFPGA platform. Section 3 presents the basic design of a virtualized data plane based on programmable hardware; this design is agnostic to any specific programmable hardware platform. Section 4 presents an implementation of our design using the NetFPGA platform. Section 5 presents a preliminary evaluation of the platform's performance; Section 6 discusses limitations and possible extensions of the current implementation. Section 7 concludes with a summary and discussion of future work.

2. Background and Related Work

This section offers background on programmable hardware technologies in networking, as well as various attempts to design and implement virtual routers.

Generally, there are three options for high-speed packet processing: Application Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs), and Network Processors. ASICs are typically customized for a specific application, such as desktop processors or graphical processing units. ASICs can host many transistors on a single chip and run at higher speeds, but development cycles are slower, and manufacturing is more complicated, resulting in high, non-recurring engineering cost. FPGAs trade complexity and performance for flexibility: they can be programmed for a specific purpose after manufacturing. FPGAs can achieve clock speeds up to 500 MHz, sufficient for many high-end applications including switches and routers that require processing of hundreds of thousands of flows per second. Network processors have feature sets that are specifically designed for networking applications and are used in routers, switches, firewalls, and intrusion detection systems. Network processors are also programmable and provide generic functions that can be used for specific networking applications. There are not many examples of high-speed network virtualization solutions. SPP [17] is one; it uses Intel IXP [11] network processors for virtualization. Network processors are quite powerful, but with the change of network processor vendor or drop of support from vendor for a particular network processor, maintenance of specific implementations becomes difficult. Similarly, any change in the vendor provided network processor programming framework is also challenging, in terms of network application maintenance.

Juniper E-series routers provide support for virtual routers [12]. These E-Series routers can support up to 1,000 forwarding tables, but there are no implementation details about these virtual routers, such as whether they use tunneling or encapsulation, or whether they rewrite packet headers or there is some other technique being used. In contrast, we are developing an hardware-based open virtualized data plane that can be used on *commodity* platforms.

NetFPGA [13, 18] is an FPGA-based platform for packet processing. The platform uses the Xilinx FPGA and has four Gigabit Ethernet interfaces. The card can reside in the PCI or PCI-X slot of any machine and can be programmed like a normal FPGA. It has two FPGAs: the Virtex2Pro 50 that hosts the user program logic, and a SPARTAN II, which has PCI interface and communication logic. The card has 4.5 MB of SRAM, which can operate at 125 MHz; the size of DRAM is 64 MB. With a 125 MHz clock rate and a 64-bit wide data path, the FPGA can provide a theoretical throughput of 8 Gbps. Our design uses SRAM for packet storage and BRAM and SRL16e storage for forwarding information and uses the PCI interface to send or receive packets from the host machine operating system. The NetFPGA project provides reference implementations for various capabilities, such as the ability to push the Linux routing table to the hardware. Our framework extends this implementation.

3. Design Goals

This section outlines our design goals for a hardware-based virtual data plane, as well as the challenges with achieving each of these design goals.

1. **Virtualization at layer two.** Many alternative protocols to IP, like Accountable Internet Protocol (AIP) [1], Host Identity Protocol (HIP) [15], and Locator Identifier Separation Protocol (LISP) [9] exist, but there is no way to understand the behavior and performance of these protocols at higher speeds. Experimenters and service providers may wish to build virtual networks that run these or some other protocols besides IP at layer three. Therefore, we aim to facilitate virtual networks that provide the appearance of layer-two connectivity between virtual nodes. Alternatives for achieving this design goal, are tunneling/encapsulation, rewriting packet headers, or redirecting packets based on virtual MAC addresses. In Section 4, we justify our decision to use redirection.

2. **Fast forwarding.** The infrastructure should forward packets as quickly as possible. To achieve this goal, we push each virtual node's forwarding tables to hardware, so that the interface card itself can forward packets on behalf of the virtual node. Forwarding packets directly in hardware, rather than passing each packet up to a software routing table in the virtual context, results in significantly faster forwarding rates, less latency, and higher throughput. The alternative—copying packets from the card to the host operating system—requires servicing interrupts, copying packets to memory, and processing the packet in software, which is significantly slower than performing the same set of operations in hardware.
3. **Resource guarantees per virtual network.** The virtualization infrastructure should be able to allocate specific resources (bandwidth, memory) to specific virtual networks. Providing such guarantees in software can be difficult; in contrast, providing hard resource guarantees in hardware is easier. Given that the hardware forwarding infrastructure has a fixed number of physical interfaces, however, the infrastructure must determine how to divide resources across the virtual interfaces that are dedicated to a single physical interface.

The next section describes the hardware architecture that allows us to achieve these goals.

4. Design and Preliminary Implementation

This section describes a design and preliminary implementation of a hardware-based virtual data plane. The system associates each incoming packet with a virtual environment and forwarding table. In contrast to previous work, the hardware itself makes forwarding decisions based on the packet's association to a virtual forwarding environment; this design provides fast, hardware-based forwarding for up to eight virtual routers running in parallel on shared physical hardware. The number of virtual routers is limited by on-chip memory usage; ultimately, placing forwarding tables off-chip may significantly increase the number of virtual routers. By separating the control plane for each virtual node (*i.e.*, the routing protocols that compute paths) from the data plane (*i.e.*, the infrastructure responsible for forwarding packets), each virtual node can have a separate control plane, independent of the data-plane implementation.

Overview In our current implementation, each virtual environment can have up to four virtual ports; this is a characteristic of our current NetFPGA-based implementation, not a fundamental limitation of the design itself. The physical router has four output ports and, hence, four output queues. Each virtual MAC is associated with one output queue at any time; this association is not fixed and changes with each incoming packet. Increasing the number of output queues allows us to increase the number of virtual ports per virtual router. The maximum number of virtual ports then depends on the resources that are available for the output queues. In addition to more output queues, we also need to increase the size of VMAC-VE (Virtual MAC to Virtual Environment) mapping table and the number of context registers associated with a particular instance of virtual router. There are four context registers for each virtual router. These registers help to add source MAC addresses for each outgoing packet, depending upon the outgoing port of packet.

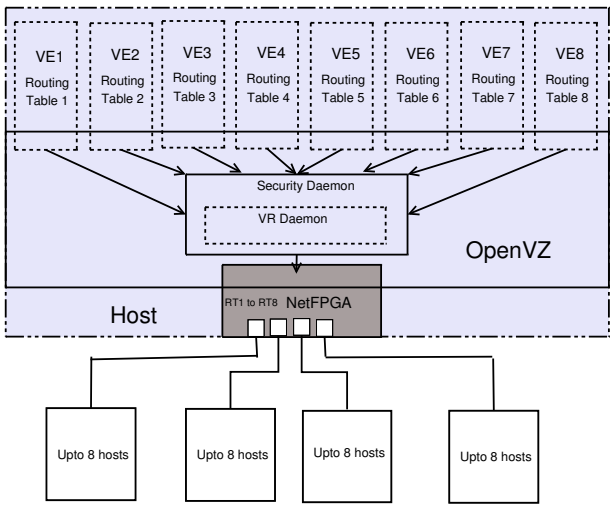

Figure 1: OpenVZ and virtual router.

Each virtual port on the NetFPGA redirects the packet to the appropriate virtual environment or forwards the packet to the next node, depending on the destination address and the packet's association to a particular virtual environment. We achieve this association by establishing a table that maps virtual MAC addresses to virtual environment. These virtual MAC addresses are assigned by the virtual environment owner and can be changed at any time. By doing so, the system can map traffic from virtual links to the appropriate virtual environments without any tunneling.

In the remainder of this section, we describe the system architecture. First, we describe the control plane, which allows router users to install forwarding table entries into the hardware, and how the system controls each virtual environment's access to the hardware. Next, we describe the software interface between processes in each virtual environment and the hardware. Finally, we describe the system's data plane, which multiplexes each packet into the appropriate virtual environment based on its MAC address.

4.1 Control Plane

The virtual environment contains two contexts: the virtual environment context (the "router user") and the root context (the "super user"). The router user has access to the container that runs on the host machine. The super user can control all of the virtual routers that are hosted on the FPGA, while router users can only use the resources which are allocated to them by the super user. Our virtual router implementation has a set of registers in FPGA that provides access to the super user and to the router users. This separation of privilege corresponds to what exists in a typical OpenVZ setup, where multiple containers co-exist on a single physical machine, but only the user in the root context has access to super user privileges.

To install forwarding-table entries, different routing protocols can be used that compute path and add these entries to forwarding table. Virtual environment allows users to run these protocols and update forwarding table entries according to their own protocol's calculations. For this update, selection of the forwarding table in hardware is done using the control register while a router user's access to that forwarding table is controlled using VMAC-VE table.

Virtual environments As in previous work (*e.g.*, Trellis [5]), we virtualize the control plane by running multiple virtual environments on the host machine. The number of OpenVZ environments is independent of the virtual routers sitting on FPGA, but the hardware can support at most eight virtual containers; *i.e.*, only eight virtual environments can have fast-path forwarding using the FPGA at one time. Each container has a router user, which is the root user for the container; the host operating system's root user has super user access to the virtual router. Router users can use a command-line based tool to interact with their instance of a virtual router. These users can read and write the routing table entries and specify their own context register values.

All the update/read requests for routing table entries must pass through the security daemon and the virtual router daemon, as shown in Figure 1. The MAC addresses stored in the context registers must be the same addresses that the virtual router container uses to reply to ARP requests. Once a virtual router user specifies the virtual port MAC addresses, the super user enters these addresses in the VMAC-VE table; this mechanism prevents a user from changing MAC addresses arbitrarily.

Hardware access control This *VMAC-VE table* stores all of the virtual environment ID numbers and their corresponding MAC addresses. Initially, this table is empty but provides access to a virtual router for a virtual environment user. The system provides a mechanism for mediating router users' access to the hardware resources. The super user can modify the VMAC-VE (Virtual MAC and Virtual Environment mapping) table. The super user grants the router user access to the fast-path forwarding provided by the hardware virtual router by adding the virtual environment ID and the corresponding MAC addresses to the VMAC-VE table. If the super user wants to destroy a virtual router or deny some users access to the forwarding table, it simply removes the virtual environment ID of the user and its corresponding MAC addresses. Access to this VMAC-VE table is provided by a register file, which is only accessible to super user.

Control register As shown in Figure 1, each virtual environment copies the routing table from its virtual environment to shared hardware. A 32-bit control register stores the virtual environment ID. When a virtual environment needs to update its routing tables, it sends its request to the *virtual router daemon* via the security daemon. After verifying the virtual environment's permissions, this daemon uses the control register to select routing tables that belong to the requesting virtual environment and updates the IP lookup and ARP table entries for that virtual environment. After updating the table, the daemon resets the control register value.

4.2 Software Interface

As shown in Figure 1, the *security daemon* prevents unauthorized changes to the routing tables by controlling access to the virtual router control register. The virtual router control register selects the virtual router for forwarding table updates. The security daemon exposes an API that router users can use to interact with their respective routers, including reading or writing the routing table entries. Apart from providing secure access to all virtual router users, the security daemon logs user requests to enable auditing.

The hardware-based fast path cannot process packets with IP options or ARP packets. These packets are sent to the virtual router daemon without any modifications. The virtual router daemon also maintains a copy of the VMAC-VE table. It examines at the packet's destination MAC and sends the packet to the corresponding virtual environment on the host machine. Similarly, when the packet is sent from any of the containers, it is first received by the virtual router daemon through the security daemon, which sends the packet to the respective virtual router in hardware for forwarding.

The super user can interact with all virtual routers via a command-line interface. In addition to controlling router user accesses by changing the VMAC-VE table, the super user can examine and modify any router user's routing table entries using the control register.

4.3 Data Plane

To virtualize the data plane in a single physical router, the router must associate each packet with its respective virtual environment. To determine a packet's association with a particular virtual environment, the router uses the virtual environment's MAC address; in addition to allowing or denying access to the virtual router users, the VMAC-VE table determines how to forward packets to the appropriate virtual environment, as well as whether to forward or drop the packet.

Mapping virtual MACs to destination VEs Once the table is populated and a new packet arrives at the virtual router, its destination MAC is looked up in the VMAC-VE table, which provides a mapping between the virtual MAC addresses and virtual environment IDs. Virtual MAC addresses in the VMAC-VE table correspond to the MAC addresses of the virtual ethernet interfaces used by virtual environment. A user has access to four registers, which can be used to update the MAC address of the user's choice. These MAC addresses must be the same as the MAC addresses of virtual environment. Since there are four ports on NetFPGA card, each virtual environment has a maximum of four MAC addresses inside this table; this is a limitation of our current implementation. As explained earlier, increasing the number of output queues and context registers will permit each virtual environment to have more than four MAC addresses. The system uses a CAM-based lookup mechanism to implement the VMAC-VE table. This design choice makes the implementation independent of any particular vendor's proprietary technology. For example, the proprietary TEMAC core from Xilinx provides a MAC address filtering mechanism, but it can only support 4 to 5 MAC addresses per TEMAC core, and most importantly it cannot demultiplex the incoming packets to the respective VE.

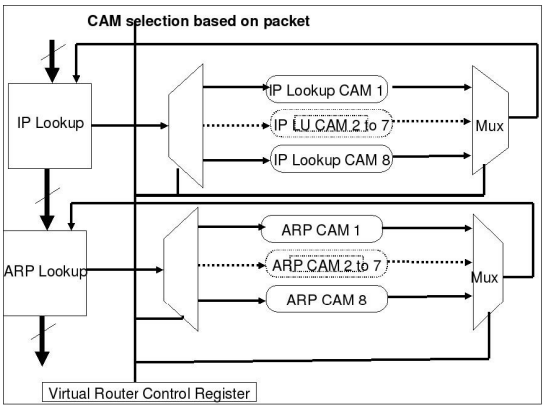

Figure 2: Virtual router table mappings.

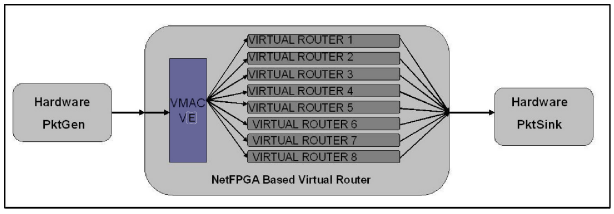

Figure 3: Experimental setup for virtual router.

Packet demultiplexing and forwarding All four physical Ethernet ports of the router are set into promiscuous mode, which allows the interface to receive any packet for any destination. After receiving the packet, its destination MAC address is extracted inside the virtual router lookup module. If there is a match in the table, the packet is processed and forwarded; otherwise, it is dropped.

This table lookup also provides the virtual environment ID (VE-ID) that switches router context for the packet that has just been received. In a context switch, all four MAC addresses of the router are changed to the MAC addresses of the virtual environment's MAC addresses. As shown in Figure 2, the VE-ID indicates the respective IP lookup module. In the case of IP lookup hit, the MAC address of next hop's IP is looked up in ARP lookup table. Once the MAC address is found for the next-hop IP, the router must provide the source MAC address for the outgoing packet. Then, context registers append the corresponding source MAC address and send the packet.

Based on the packet's association with a VE, the context register values are changed to correspond to the four MAC addresses for the virtual router in use. The router's context remains active for the duration of a packet's traversal through the FPGA and changes when the next incoming packet arrives. For a packet, each virtual port appears as one physical port with its own MAC address. Once the forwarding engine decides a packet's fate, it is directed to the appropriate output port. The outgoing packet must have the source MAC address that corresponds to the virtual port that sends the packet. To provide each packet with its correct source MAC address, the router uses context registers. The number of context registers is equal to the number of virtual ports associated with the particular router. The current implementation uses four registers, but this number can be increased if the virtual router can support more virtual ports.

Shared functions Our design shares resources between virtual routers on the same FPGA. It only replicates the resources that are necessary to implement fast path forwarding. To understand the virtual router context and its switching when new packets arrive, we describe the modules that can be shared in an actual router and modules that cannot be shared. Router modules that involve decoding of packets, calculating checksums, decrementing TTLs, etc. can be shared between different routers, as they do not maintain any state that is specific to a virtual environment. Similarly, the input queues and input arbiter is shared between the eight virtual routers. Packets belonging to any virtual router can come into any of the input queues where the arbiter feeds these packets to the virtual router lookup module. Output queues are shared between different virtual routers, and packets from different virtual routers can be placed into any output queue.

VE-specific functions Some resources cannot be shared between the routers. The most obvious among them is the forwarding information base. In our current virtual router implementation, we have used NetFPGA's reference router implementation as our base implementation. In this implementation a packet that needs to be forwarded, needs at least three information resources namely IP lookup table, MAC address resolution table and router's MAC addresses. These three resources are unique to every router instance and they can not be removed and populated back again with every new packet. Therefore, the implementation maintains a copy of each of these resources for every virtual router.

5. Results

This section presents the results of our preliminary evaluation. First, we evaluate the performance of the system; we then discuss the implementation complexity for the virtual router implementation (in terms of gates), compared to its base router implementation. We show that our hardware-based implementation provides comparable packet forwarding rates to Linux kernel packet forwarding. We then show the scalability of the implementation in terms of forwarding rate by comparing the forwarding rate of fixed-sized packets with different numbers of virtual routers. We benchmark our implementation against the base router implementation. The system also provides isolation to the host resources. (At this point, the system provides no bandwidth guarantees among different virtual routers sitting on same FPGA card.) Finally, we show that gate count for hosting multiple virtual routers is 75% less than the hardware required to host an equivalent number of physical routers.

We used the NetFPGA-enabled routers in the Emulab testbed [8] to evaluate the performance of our system. We used a simple source, router, and sink topology. Figure 3 shows the experimental setup for the virtual router. For the source and sink, we used a hardware packet generator and receivers. The host machine for the Linux router had 3 GHz

single processor, 2 GB RAM, two 1 Gbps interfaces. We sent a mix of traffic belonging to each of the different virtual networks and sent the traffic to the virtual router accordingly. Depending on packet's association, the hardware forwards the packet to the appropriate virtual router, which in turn sends the traffic to the appropriate output queue.

5.1 Performance

We evaluate performance according to three requirements: (1) forwarding rate, (2) scalability, and (3) isolation. We discuss how the software-based definition of isolation changes in the context of programmable hardware routers, as well as how scalability is affected by limited hardware resources.

5.1.1 *Packet forwarding rate*

Previous work has measured the maximum sustainable packet forwarding rate for different configurations of software-based virtual routers. We also measure packet forwarding rates and show that hardware-accelerated forwarding can increase packet forwarding rates without incurring any cost for host machine resources. To compare the forwarding rates between hardware and software, we compare forwarding rates of Linux and NetFPGA-based router implementations from the NetFPGA group, as shown in Figure 4. We show the maximum packet forwarding rate that can be achieved for each configuration. The maximum forwarding rate shown, about 1.4 million packets per second, is the maximum traffic rate which we were able to generate through the NetFPGA-based packet generator.

The Linux kernel drops packets at high loads, but our configuration could not send packets at a high enough rate to induce packet drops in hardware. If we impose the condition that zero packets should be dropped at the router, then the packet forwarding rates for Linux router drops significantly, but the forwarding rates for hardware based router remain constant. Figure 4 shows packet forwarding rates when this "no packet drop" condition is *not* imposed (*i.e.*, we measure the maximum sustainable forwarding rates).

For large packets, we were able to achieve the same forwarding rate using in-kernel forwarding as using a single port of NetFPGA router. Once the packet size drops below 200 bytes, however, the software-based router is unable to keep up with the forwarding requirements. The native Linux forwarding rates look decent in our experimental setup for larger packet sizes, but increasing the number of interfaces will result in more significant differences. The NetFPGA router can sustain up to 4 Gbps without incurring any CPU usage of host machine. Since our current setup had only two 1 Gbps network interface cards installed, we could test for 1Gbps rates only; in ongoing work, we are testing the card with higher traffic rates. To properly compare the forwarding rates with native Linux forwarding, we must add four NICs to compare performance with NetFPGA hardware router; in such a configuration, we expect the forwarding rates to drop significantly.

As expected, hardware-based packet forwarding is significantly faster than the raw kernel-based packet forwarding. The packet forwarding rate starts dropping with an increase in packet size. This drop is understandable, since the input ports cannot forward traffic greater than 1 Gbps. To compare

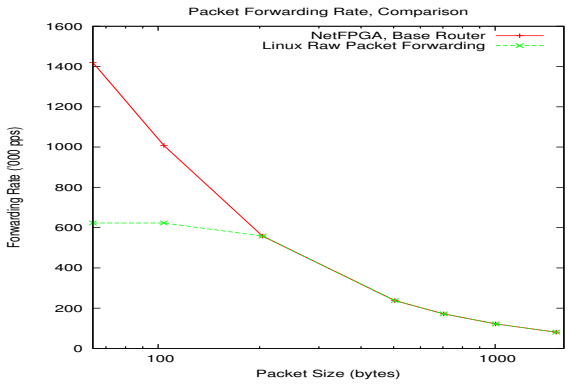

Figure 4: Comparison of forwarding rates.

the overhead of virtualizing the hardware-based platform, we compared the packet-forwarding rate for a single virtual router with that of the hardware-based packet forwarder and found that the forwarding rate of the hardware-based virtual router was the same. Because the base router has a pipelined architecture, adding another module consumes more resources, but it has no effect on forwarding throughput. The VMAC-VE table is implemented in CAM; it accepts and rejects packets in one stage, while the next module demultiplexes the packets based on information forwarded by virtual router lookup module. Therefore, there is no effect on forwarding rate for either implementation.

5.1.2 *Scalability*

Hardware-based virtual routers exploit the inherent parallelism in hardware and avoid context-switching overhead. In a software-based virtual router, a context switch wastes CPU cycles for saving and restoring packet context. Because hardware naturally lends itself to this parallelism, the next challenge was to come up with a design that is scalable and should not impose a performance hit on packet forwarding rates. Our current implementation has eight virtual routers on one FPGA card; the number of virtual routers is only limited by the hardware resources available on the Virtex-II Pro 50 FPGA card. (Section 5.2 discusses these trends.)

With each incoming packet, the router must change its context based on the MAC address of the incoming packet. A scalable virtual router implementation should be able to switch contexts and forward packets with minimal effects on forwarding rate. To observe virtual router switching overhead in our implementation, we compared packet forwarding rates with different number of virtual routers. We used fixed-size packets and increased the number of virtual router from one to eight. We used the minimum packet size, 64 bytes, to provide least amount of time for context switching. We found that increasing the number of virtual routers had no effect on packet-forwarding rate.

5.1.3 *Isolation*

Forwarding packets in hardware provides relatively better resource isolation then host machine-based virtual router solution (although control plane traffic does not completely isolate CPU resources). To measure CPU isolation, we used

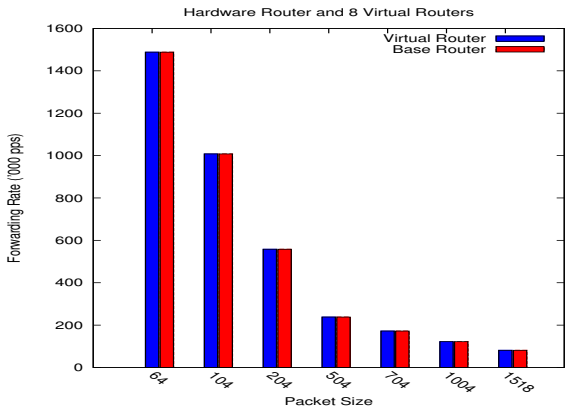

Figure 5: Data plane performance: virtual vs. reference router.

eight virtual routers to forward traffic when the host CPU was utilized at 100% by a user space process. We then sent traffic where each user had an assigned traffic quota in packets per second. When no user exceeds the assigned quotas, the router forwards traffic according to the assigned rates with no packet loss. Our design still lacks several isolation-related features, which we discuss in Section 6.

To study the behavior of the virtual router under congestion and to evaluate its isolation properties, we set up a topology where two physical 1 Gbps ports of routers were flooded at 1 Gbps and a sink node was connected to a third physical 1 Gbps port. To load the router, we used eight virtual routers to forward traffic to the same output port. When two router input ports are flooded using hardware-based packet generators, the router receives packets at 2 Gbps and forwards all packets to the corresponding output port. At the output port, queued packets are dropped at random; the router forwards at a rate of 1 Gbps. This behavior is the same while flooding one output port of the physical router, as well as in the virtual router setup when all eight virtual routers are forwarding traffic to one output port.

In a scenario where a particular virtual router forwards traffic at a rate that is beyond a limit allowed by policy, the router can enforce the limit at either the ingress or egress port. To enforce this policy, the super user could monitor forwarding rates and periodically remove VMAC-VE table entries to start dropping user's packets, depending on allocations and sustained forwarding rates.

5.2 Resource Utilization

We evaluated the complexity of our design in terms of its resource utilization on the NetFPGA platform. We measure both logic and memory utilization and compare this utilization to that of the NetFPGA reference router implementation. We evaluated both the reference implementation and our design on the Xilinx Virtex-II Pro 50 FPGA. The reference design has a 32-entry TCAM longest prefix match module with its lookup table, and a 32-entry ARP table. The lookup module is implemented using SRL16e while ARP CAM is implemented using dual port Block RAM (BRAM) [6]. Our virtual router implementation, on the other hand, uses a 32-entry CAM for the VMAC-VE lookup table that is based on BRAM. There are a total of eight TCAMs with their lookup tables, and eight ARP tables, each with 32 entries. We aim for small lookup tables, each with the same capacity as our physical router design.

We measured our results using Xilinx ISE 9.2i and show our results as the total resource utilization of the FPGA. Our current implementation does not include any modification to the routers' transmit or receive queues, but providing fine-grained control for virtual routers will require modification to these queues; we discuss these modifications in Section 6.

LUT and BRAM Utilization The base router implementation requires a total of 23,330 four-input LUTs, which consume about 45% of the available logic on the NetFPGA. The implementation also requires 123 BRAM units, which is 53% of available BRAM.

The virtual router implementation uses 32,740 four-input LUTs, which account for approximately 69% of LUTs devoted to logic implementation. Of this 69%, approximately 13% of LUTs are used for shift registers. Route through accounts for 4.5%, with a total of 73% of LUTs that are used in current design. This implementation uses 202 BlockRAMs, or 87% of available BRAM.

Gate count Another measure of complexity is equivalence to total number of logic gates, based on estimates from Xilinx tools. Although we do not have any numbers about gate count usage of commercial routers, we are not aware of any open-source router implementation other then NetFPGA-based reference router implementation, so we can only compare virtual router implementation with NetFPGA group's router implementation.

The current NetFPGA router design provides line-rate forwarding and has one IPv4 router implemented; this implementation is equivalent to about 8.6 million gates. In contrast, our virtual router implementation, which provides line-rate forwarding and can have up to eight IPv4 virtual routers to a total of 14.1 million logic gates, *considerably less* than the total equivalent gate cost for eight physical routers. (Based on statistics reported by Xilinx ISE 9.2, the gate cost for eight physical routers will be around 70 million logic gates, assuming the NetFPGA-based design.) Such an implementation may provide more bandwidth, but will also require more power. In summary, our virtual router design provides eight virtual routers at approximately 21% to 25% of the gate cost of today's cutting edge router technology, with correspondingly less power usage.

6. Limitations and Future Work

In this section, we discuss several limitations and possible extensions to the current implementation. Some of these limitations result from the need for virtual routers to behave like a physical router so that they can be managed separately by individual users. Some limitations arise from the need to provide flexibility in virtual networks. We are extending the current design to address these limitations.

First, providing individual router users with statistics about their networks would help with various accounting tasks. The virtual data plane we have presented could be extended to collect statistics about traffic in each virtual net-

work. Second, the current implementation can support only small forwarding tables, as it uses BRAM for table storage. This design can be improved by storing forwarding tables in SRAM to make this design capable of supporting large forwarding tables. Third, virtual routers should allow users to update the forwarding tables without interrupting packet forwarding. Our current implementation lacks this feature: when one user tries to update the forwarding table, the others are blocked. Allowing concurrent packet forwarding and forwarding-table updates also requires a different register-set interface for each virtual router.

Finally, each virtual router should operate independently of other virtual routers on the device. As shown in Section 5, the current design meets requirements for fast forwarding and scalability and provides isolation to the CPU running on the host machine of the NetFPGA card. However, the current implementation does not isolate the traffic between different virtual networks: in the current implementation, all virtual networks share the same physical queue for a particular physical interface, so traffic in one virtual network can interfere with the performance observed by a different virtual network. In the ideal case, no traffic in one virtual network should affect other's bandwidth. In our ongoing work, we are designing various techniques to provide isolation, while still making efficient use of the relatively limited available resources.

7. Conclusion

Sharing the same physical substrate among a number of different virtual networks amortizes the cost of the physical network; as such, virtualization is promising for many networked applications and services. To date, however, virtual networks typically provide only software-based support for packet forwarding, which results in both poor performance and isolation. The advent of programmable network hardware has made it possible to achieve improved isolation and packet forwarding rates for virtual networks; the challenge, however, is designing a hardware platform that permits sharing of common hardware functions across virtual routers without compromising performance or isolation.

As a first step towards this goal, this paper has presented a design for a fast, virtualized data plane based on programmable network hardware. Our current implementation achieves the isolation and performance of native hardware forwarding and shares hardware modules that are common across virtual routers, which achieves up to 75% savings in the overall amount of logic that is required to implement independent physical routers. Although many more functions can ultimately be added to such a hardware substrate (*e.g.*, enforcing per-virtual router resource constraints), we believe our design represents an important first step towards the ultimate goal of supporting a fast, programmable, and scalable hardware-based data plane for virtual networks.

Acknowledgments

This work was funded by NSF CAREER Award CNS-0643974 and NSF Award CNS-0626950. We thank Ramki Gummadi, Eric Keller, and Murtaza Motiwala for helpful feedback and suggestions.

REFERENCES

[1] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proc. ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[3] 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. First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[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, Aug. 2006.

[5] S. Bhatia, M. Motiwala, W. Muhlbauer, Y. Mundada, V. Valancius, A. Bavior, N. Feamster, L. Peterson, and J. Rexford. Trellis: A Platform for Building Flexible, Fast Virtual Networks on Commodity Hardware. In *3rd International Workshop on Real Overlays & Distributed Systems*, Oct. 2008.

[6] J. Brelet and L. Gopalakrishnan. Using Virtex-II Block RAM for High Performance Read/Write CAMs. `http://www.xilinx.com/support/documentation/application_notes/xapp260.pdf`.

[7] Cisco Multi-Topology Routing. `http://www.cisco.com/en/US/products/ps6922/products_feature_guide09186a00807c64b8.html`.

[8] Emulab. `http://www.emulab.net/`.

[9] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/ID Separation Protocol (LISP). RFC DRAFT-12, Internet Engineering Task Force, Mar. 2009.

[10] N. Feamster, L. Gao, and J. Rexford. How to lease the Internet in your spare time. *ACM Computer Communications Review*, 37(1):61–64, 2007.

[11] Intel IXP 2xxx Network Processors. `http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm`.

[12] JunOS Manual: Configuring Virtual Routers. `http://www.juniper.net/techpubs/software/erx/junose72/swconfig-system-basics/html/virtual-router-config5.html`.

[13] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education*, pages 160–161. IEEE Computer Society Washington, DC, USA, 2007.

[14] Juniper Networks: Intelligent Logical Router Service. `http://www.juniper.net/solutions/literature/white_papers/200097.pdf`.

[15] R. Moskowitz and P. Nikander. Host identity protocol (hip) architecture. RFC 4423, Internet Engineering Task Force, May 2006.

[16] OpenVZ: Server Virtualization Open Source Project. `http://www.openvz.org`.

[17] J. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, et al. Supercharging PlanetLab: A High Performance, Multi-application, Overlay Network Platform. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.

[18] G. Watson, N. McKeown, and M. Casado. NetFPGA: A Tool for Network Research and Education. In *2nd workshop on Architectural Research using FPGA Platforms (WARFP)*, 2006.