

# A Fast, Virtualized Data Plane for the NetFPGA

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.

## 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 containers like OpenVZ). Creating virtual links involves creating tunnels between these virtual nodes (*e.g.*, with Ethernet-based GRE tunneling [4]). This technology potentially enables multiple service providers to share the cost of physical infrastructure. Major router vendors have begun to embrace router virtualization [5,7,9], and the research community has followed suit in building support for both virtual network infrastructures [2–4, 11] and services that could run on top of this infrastructure [6].

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 strong isolation. The advent of programmable network hardware (*e.g.*, NetFPGA [8, 12]), 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 a large number of 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 [10], Xen [1]); we virtualize the data plane by multiplexing the resources on the hardware itself.

One of the major challenges in designing a hardware-based platform for a virtualized data plane 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 forwarding 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 the design, implementation, and preliminary 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 also 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. Additionally, our 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.

The rest of this paper is organized as follows. Section 2 presents the basic design of a virtualized data plane based on

programmable hardware; this design is agnostic to any specific programmable hardware platform. Section 3 presents an implementation of our design using the NetFPGA platform. Section 4 concludes with a summary and discussion of future work.

## 2. 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.** Experimenters and service providers may wish to build virtual networks that run other protocols besides IP at layer three. Therefore, we aim to facilitate virtual networks that provide the appearance of layer-two connectivity between each virtual node. This function provides the illusion of point-to-point connectivity between pairs of virtual nodes. Alternatives, for achieving this design goal, are tunneling/encapsulation, rewriting packet headers, or redirecting packets based on virtual MAC addresses. In Section 3, we justify our design 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 copying packets to memory, servicing interrupts, 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, since each virtual network can simply receive a fixed number of clock cycles. Given that the hardware forwarding infrastructure has a fixed number of physical interfaces, however, the infrastructure must also 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.

## 3. Design and Implementation

This section describes our design and 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. 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 how much resources we have to allocate for the output queues. In addition to more output queues we would 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 and they are used to add source MAC addresses for each outgoing packet, depending upon the outgoing port of packet.

Each virtual port on the NetFPGA redirects the packet to the appropriate virtual environment or forward 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 the addresses assigned by the virtual environment owner and can be changed 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 path, which multiplexes each packet into the appropriate virtual environment based on its MAC address.

### 3.1 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 that which exists in a typical OpenVZ setup, where multiple containers co-exist on a sin-

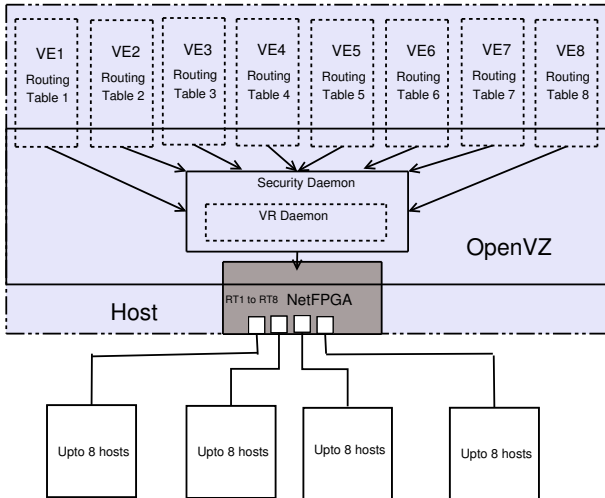


Figure 1: OpenVZ and virtual router.

gle physical machine, but only the user in the root context has access to super user privileges.

**Virtual environments** As in previous work (e.g., Trelis [4]), 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. 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 virtual router. These users can read and write the routing table entries and specify their own context register values.

Each virtual router user can run any routing protocol, but all the routing table entries update/read requests 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 for the 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 to provide access of a virtual router to 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. 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 environ-

ment 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 that is currently being controlled. Whenever a virtual environment needs to update its routing tables, it sends its request to *virtual router 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 particular virtual environment. After updating the table values, daemon resets the control register value to zero.

### 3.2 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 is used to select 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 *software interface* provides a mechanism for processing packets using software exceptions. The hardware-based fast path cannot process packets with the IP options or ARP packets, for example. These packets are sent to virtual router daemon without any modifications, virtual router daemon, also maintains a copy of VMAC-VE table. It looks at the packet’s destination MAC and sends the packet to the corresponding virtual environment running on the host environment. Similarly, when the packet is sent from any of the containers, it first received by 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. Apart from 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 control register.

### 3.3 Data Plane

To provide virtualization 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 virtual environment’s MAC address; this MAC address as described earlier, apart from allowing/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 VMAC-VE table, which provides mapping between the virtual MAC addresses and

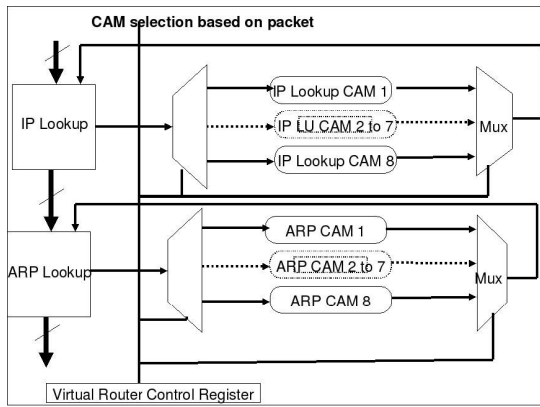


Figure 2: Virtual router table mappings.

virtual environment IDs. Virtual MAC addresses in 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 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 only 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 can't provide demuxing of the incoming packets to the respective VE.

**Packet demultiplexing and forwarding** In the current implementation, all four physical ethernet ports of the router are set into promiscuous mode, which allows the interface receive any packet for any destination. After receiving the packet, its destination MAC address is extracted inside the virtual router lookup module, as shown in Figure 2. If there is a match in the table, the packet processed and forwarded; otherwise, it is dropped.

This table lookup also provides the virtual environment ID (VE-ID) that is used to switch router context for the packet which 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 3, 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 needs to provide the source MAC address for the outgoing packet. Then, context registers are used to 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 that correspond to the four

MAC addresses for virtual router in use. The router's context remains active for the duration of a packet's traversal through FPGA and changes when the next incoming packet arrives. Each virtual port appears 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 maximizes available resources to share different resources with other routers on the same FPGA. It only replicates those resources which are really necessary to implement fast path forwarding. To understand virtual router context and its switching with every new packet, we first 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 and they are picked up by arbiter to be fed into virtual router lookup module. Output queues are shared between different virtual routers, and packets from different virtual routers can be placed in any output queue.

**VE-specific functions** Some resources can not 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 architecture maintains a copy of each of these resources for every virtual router. The current implementation maintains a separate copy of all these resources for every virtual router instantiated inside the FPGA.

## 4. Discussion and Future Work

In this section, we describe several possible extensions to the current implementation. Some of these additions come from making these virtual routers more router like; some stem from the requirements of extending the current implementation to better support virtual networks. We believe that the current implementation must have a minimum set of these additions to completely function as a virtual router.

The virtualized data plane we have presented could be extended to support collecting statistics about network traffic in each virtual network. Today, administrators of physical networks can obtain the traffic statistics for their respective

networks; we aim to provide similar function for virtual networks. Extending this function to the NetFPGA requires adding new function to the current logic on the NetFPGA; it also entails addressing challenges regarding the efficient use of the relatively limited memory available on the physical card itself.

Each physical router user is able to update her forwarding tables with the frequency of their like. Our current implementation lacks this feature as once one user tries to update his/her respective table others are blocked. Allowing concurrent packet forwarding and forwarding-table updates requires a completely different register set interface for each virtual router to update its respective forwarding table.

Virtual routers must provide speed, scalability and isolation. We have been able to meet the fast forwarding path and scalability requirements. The hardware-based implementation 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 an ideal case, no traffic in a different virtual network should affect other virtual router's bandwidth. In our ongoing work, we are examining how various queuing and scheduling disciplines might be able to provide this type of isolation, while still making efficient use of the relatively limited available resources.

## 5. 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 implements shares hardware modules that are common across virtual 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.

## REFERENCES

- [1] 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.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] Cisco Multi-Topology Routing. [http://www.cisco.com/en/US/products/ps6922/products\\_feature\\_guide09186a00807c64b8.html](http://www.cisco.com/en/US/products/ps6922/products_feature_guide09186a00807c64b8.html).
- [6] 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.
- [7] JunOS Manual: Configuring Virtual Routers. <http://www.juniper.net/techpubs/software/erx/junose72/swconfig-system-basics/html/virtual-router-config5.html>.
- [8] 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.
- [9] 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).
- [10] OpenVZ: Server Virtualization Open Source Project. <http://www.openvz.org>.
- [11] 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.
- [12] 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.

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of