SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware

Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, Nick Feamster
School of Computer Science, Georgia Tech

ABSTRACT

We present SwitchBlade, a platform for rapidly deploying custom protocols on programmable hardware. SwitchBlade uses a pipeline-based design that allows individual hardware modules to be enabled or disabled on the fly, integrates software exception handling, and provides support for forwarding based on custom header fields. SwitchBlade's ease of programmability and wire-speed performance enables rapid prototyping of custom data-plane functions that can be directly deployed in a production network. SwitchBlade integrates common packet-processing functions as hardware modules, enabling different protocols to use these functions without having to resynthesize hardware. SwitchBlade's customizable forwarding engine supports both longest-prefix matching in the packet header and exact matching on a hash value. SwitchBlade's software exceptions can be invoked based on either packet or flow-based rules and updated quickly at runtime, thus making it easy to integrate more flexible forwarding function into the pipeline. SwitchBlade also allows multiple custom data planes to operate in parallel on the same physical hardware, while providing complete isolation for protocols running in parallel. We implemented SwitchBlade using NetFPGA board, but SwitchBlade can be implemented with any FPGA. To demonstrate SwitchBlade's flexibility, we use SwitchBlade to implement and evaluate a variety of custom network protocols: we present instances of IPv4, IPv6, Path Splicing, and an OpenFlow switch, all running in parallel while forwarding packets at line rate.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design C.2.5 [Computer-Communication Networks]: Local and Wide-Area Networks C.2.6 [Computer-Communication Networks]: Internetworking

General Terms: Algorithms, Design, Experimentation, Performance

Keywords: Network Virtualization, NetFPGA

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM 2010, August 30-September 3, 2010, New Delhi, India.
Copyright 2010 ACM 978-1-4503-0201-2/10/08 ...$10.00.

1. INTRODUCTION

Countless next-generation networking protocols at various layers of the protocol stack require data-plane modifications. The past few years alone have seen proposals at multiple layers of the protocol stack for improving routing in data centers, improving availability, providing greater security, and so forth [3, 13, 17, 24]. These protocols must ultimately operate at acceptable speeds in production networks—perhaps even alongside one another—which raises the need for a platform that can support fast hardware implementations of these protocols running in parallel. This platform must provide mechanisms to deploy these new network protocols, header formats, and functions quickly, yet still forward traffic as quickly as possible. Unfortunately, the conventional hardware implementation and deployment path on custom ASICs incurs a long development cycle, and custom protocols may also consume precious space on the ASIC. Software-defined networking paradigms (e.g., Click [7, 16]) offer some hope for rapid prototyping and deployment, but a purely software-based approach cannot satisfy the strict performance requirements of most modern networks. The networking community needs a development and deployment platform that offers high performance, flexibility, and the possibility of rapid prototyping and deployment.

Although other platforms have recognized the need for fast, programmable routers, they stop somewhat short of providing a programmable platform for rapid prototyping on the hardware itself. Platforms that are based on network processors can achieve fast forwarding performance [22], but network processor-based implementations are difficult to port across different processor architectures, and customization can be difficult if the function that a protocol requires is not native to the network processor's instruction set. All other functions should be implemented in software. PLUG [8] is an excellent framework for implementing modular lookup modules, but the model focuses on manufacturing high-speed chips, which is costly and can have a long development cycle. RouteBricks [12] provides a high-performance router, but it is implemented entirely in software, which may introduce scalability issues; additionally, prototypes developed on RouteBricks cannot be easily ported to hardware.

This paper presents SwitchBlade, a programmable hardware platform that strikes a balance between the programmability of software and the performance of hardware, and enables rapid prototyping and deployment of new protocols. SwitchBlade enables rapid deployment of new protocols on hardware by providing modular building blocks to afford customizability and programmability that is sufficient for implementing a variety of data-plane functions. SwitchBlade's ease of programmability and wire-speed performance enables rapid prototyping of custom data-plane functions that can be directly deployed in a production network. SwitchBlade

relies on field-programmable gate arrays (FPGAs). Designing and implementing SwitchBlade poses several challenges:

- *Design and implementation of a customizable hardware pipeline.* To minimize the need for resynthesizing hardware, which can be prohibitive if multiple parties are sharing it, SwitchBlade's packet-processing pipeline includes hardware modules that implement common data-plane functions. New protocols can select a subset of these modules on the fly, without resynthesizing hardware.
- *Seamless support for software exceptions.* If custom processing elements cannot be implemented in hardware (*e.g.*, due to limited resources on the hardware, such as area on the chip), SwitchBlade must be able to invoke software routines for processing. SwitchBlade's hardware pipeline can directly invoke software exceptions on either packet or flow-based rules. The results of software processing (e.g., forwarding decisions), can be cached in hardware, making exception handling more efficient.
- *Resource isolation for simultaneous data-plane pipelines.* Multiple protocols may run in parallel on same hardware; we call each data plane a Virtual Data Plane (VDP). SwitchBlade provides each VDP with separate forwarding tables and dedicated resources. Software exceptions are the VDP that generated the exception, which makes it easier to build virtual control planes on top of SwitchBlade.
- *Hardware processing of custom, non-IP headers.* SwitchBlade provides modules to obtain appropriate fields from packet headers as input to forwarding decisions. SwitchBlade can forward packets using longest-prefix match on 32-bit header fields, an exact match on fixed length header field, or a bitmap added by custom packet preprocessing modules.

The design of SwitchBlade presents additional challenges, such as (1) dividing function between hardware and software given limited hardware resources; (2) abstracting physical ports and input/output queues; (3) achieving rate control on per-VDP basis instead of per-port basis; and (4) providing a clean interface to software.

We have implemented SwitchBlade using the NetFPGA board [2], but SwitchBlade can be implemented with any FPGA. To demonstrate SwitchBlade's flexibility, we use SwitchBlade to implement and evaluate several custom network protocols. We present instances of IPv4, IPv6, Path Splicing, and an OpenFlow switch, all of which can run in parallel and forward packets at line rate; each of these implementations required only modest additional development effort. SwitchBlade also provides seamless integration with software handlers implemented using Click [16], and with router slices running in OpenVZ containers [20]. Our evaluation shows that SwitchBlade can forward traffic for custom data planes—including non-IP protocols—at hardware forwarding rates. SwitchBlade can also forward traffic for multiple distinct custom data planes in parallel, providing resource isolation for each. An implementation of SwitchBlade on the NetFPGA platform for four parallel data planes fits easily on today's NetFPGA platform; hardware trends will improve this capacity in the future. SwitchBlade can support additional VDPs with less than a linear increase in resource use, so the design will scale as FPGA capacity continues to increase.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 explains our design goals and the key resulting design decisions. Section 4 explains the SwitchBlade design, and Section 5 describes the implementation of SwitchBlade, as well as our implementations of three custom data planes on SwitchBlade. Section 6 presents performance results. Section 7 briefly describes how we have implemented a virtual router on top of SwitchBlade using OpenVZ. We discuss various extensions in Section 8 and conclude in Section 9.

2. RELATED WORK

We survey related work on programmable data planes in both software and hardware.

The Click [16] modular router allows easy, rapid development of custom protocols and packet forwarding operations in software; kernel-based packet forwarding can operate at high speeds but cannot keep up with hardware for small packet sizes. An off-the-shelf NetFPGA-based router can forward traffic at 4 Gbps; this forwarding speed can scale by increasing the number of NetFPGA cards, and development trends suggest that much higher rates will be possible in the near future. RouteBricks [12] uses commodity processors to achieve software-based packet processing at high speed. The design requires significant PCIe interconnection bandwidth to allow packet processing at CPUs instead of on the network cards themselves. As more network interface cards are added, and as traffic rates increase, however, some packet processing may need to be performed on the network cards themselves to keep pace with increasing line speeds and to avoid creating bottlenecks on the interconnect.

Supercharged PlanetLab (SPP) [22] is a network processor (NP)-based technology. SPP uses Intel IXP network processors [14] for data-plane packet processing. NP-based implementations are specifically bound to the respective vendor-provided platform, which can inherently limit the flexibility of data-plane implementations.

Another solution to achieve wire-speed performance is developing custom high-speed networking chips. PLUG [8] provides a programming model for manufacturing chips to perform high-speed and flexible packet lookup, but it does not provide an off-the-shelf solution. Additionally, chip manufacturing is expensive: fabrication plants are not common, and cost-effective manufacturing at third-party facilities requires critical mass of demand. Thus, this development path may only make sense for large enterprises and for protocols that have already gained broad acceptance. Chip manufacturing also has a high turnaround time and post-manufacturing verification processes which can impede development of new protocols that need small development cycle and rapid deployment.

SwitchBlade is an FPGA-based platform and can be implemented on any FPGA. Its design and implementation draws inspiration from our earlier work on designing an FPGA-based data plane for virtual routers [4]. FPGA-based designs are not tied to any single vendor, and it scales as new, faster and bigger FPGAs become available. FPGAs also provide a faster development and deployment cycle compared to chip manufacturing.

Casado *et al.* argue for simple but high-speed hardware with clean interfaces with software that facilitate independent development of protocols and network hardware [9]. They argue that complex routing decisions can be made in software and cached in hardware for high-speed processing; in some sense, SwitchBlade's caching of forwarding decisions that are handled by software exception handlers embodies this philosophy. OpenFlow [19] enables the rapid development of a variety of protocols, but the division of functions between hardware and software in SwitchBlade is quite different. Both OpenFlow and SwitchBlade provide software exceptions and caching of software decisions in hardware, but SwitchBlade also provides selectable hardware preprocessing modules that effectively moves more flexible processing to hard-

ware. SwitchBlade also easily accommodates new hardware modules, while OpenFlow does not.

SwitchBlade provides wire-speed support for parallel customized data planes, isolation between them, and their interfacing with virtualization software, which would make SwitchBlade a suitable data plane for a virtual router. OpenFlow does not directly support multiple custom data planes operating in parallel. FlowVisor [1] provides some level of virtualization but sits between the OpenFlow switch and controller, essentially requiring virtualization to occur in software.

Feature	Design Goals	Pipeline Stages
Virtual Data Plane (§ 4.2)	Parallel custom data planes	VDP selection
Customizable hardware modules (§ 4.3)	Rapid programming, customizability	Preprocessing, Forwarding
Flexible matching in forwarding (§ 4.4)	Customizability	Forwarding
Programmable software exceptions (§ 4.5)	Rapid programming, customizability	Forwarding

Table 1: SwitchBlade design features.

3. GOALS AND DESIGN DECISIONS

The primary goal of SwitchBlade is to enable *rapid development and deployment of new protocols working at wire-speed*. The three subgoals, in order of priority, are: (1) Enable rapid development and deployment of new protocols; (2) Provide customizability and programmability while maintaining wire-speed performance; and (3) Allow multiple data planes to operate in parallel, and facilitate sharing of hardware resources across those multiple data planes. In this section, we describe these design goals, their rationale, and highlight specific design choices that we made in SwitchBlade to achieve these goals.

Goal #1. Rapid development and deployment on fast hardware. Many next-generation networking protocols require data-plane modifications. Implementing these modifications entirely in software results in a slow data path that offers poor forwarding performance. As a result, these protocols cannot be evaluated at the data rates of production networks, nor can they be easily transferred to production network devices and systems.

Our goal is to provide a platform for designers to quickly deploy, test, and improve their designs with wire-speed performance. This goal influences our decision to implement SwitchBlade using FPGAs, which are programmable, provide acceptable speeds, and are not tied to specific vendors. An FPGA-based solution can allow network protocol designs to take advantage of hardware trends, as larger and faster FPGAs become available. SwitchBlade relies on programmable hardware, but incorporates software exception handling for special cases; a purely software-based solution cannot provide acceptable forwarding performance. From the hardware perspective, custom ASICs incur a long development cycle, so they do not satisfy the goal of rapid deployment. Network processors offer speed, but they do not permit hardware-level customization.

Goal #2. Customizability and programmability. New protocols often require specific customizations to the data plane. Thus, SwitchBlade must provide a platform that affords enough customization to facilitate the implementation and deployment of new protocols.

Providing customizability along with fast turnaround time for hardware-based implementations is challenging: a bare-bones FPGA is customizable, but programming from scratch has a high turnaround time. To reconcile this conflict, SwitchBlade recognizes that even custom protocols share common data-plane extensions. For example, many routing protocols might use longest prefix or exact match for forwarding, and checksum verification and update, although different protocols may use these extensions on different fields on in the packets. SwitchBlade provides a rich set of common extensions as modules and allows protocols to dynamically select any subset of modules that they need. SwitchBlade's modules are programmable and can operate on arbitrary offsets within packet headers.

For extensions that are not included in SwitchBlade, protocols can either add new modules in hardware or implement exception handlers in software. SwitchBlade provides hardware caching for forwarding decisions made by these exception handlers to reduce performance overhead.

Goal #3. Parallel custom data planes on a common hardware platform. The increasing need for data-plane customization for emerging network protocols makes it necessary to design a platform that can support the operation of several custom data planes that operate simultaneously and in parallel on the same hardware platform. SwitchBlade's design identifies functions that are common across data-plane protocols and provides those implementations shared access to the hardware logic that provides those common functions.

SwitchBlade allows customized data planes to run in parallel. Each data plane is called a Virtual Data Plane (VDP). SwitchBlade provides separate forwarding tables and virtualized interfaces to each VDP. SwitchBlade provides isolation among VDP using per-VDP rate control. VDPs may share modules, but to preserve hardware resources, shared modules are not replicated on the FPGA. SwitchBlade ensures that the data planes do not interface even though they share hardware modules.

Existing platforms satisfy some or all of these goals, but they do not address all the goals at once or with the prioritization we have outlined above. For example, SwitchBlade trades off higher customizability in hardware for easier and faster deployability by providing a well-defined but modular customizable pipeline. Similarly, while SwitchBlade provides parallel data planes, it still gives each data plane direct access to the hardware, and allows each VDP access to a common set of hardware modules. This level of sharing still allows protocol designers enough isolation to implement a variety of protocols and systems; for example, in Section 7, we will see that designers can run virtual control planes and virtual environments (e.g., OpenVZ [20], Trellis [6]) on top of SwitchBlade.

4. DESIGN

SwitchBlade has several unique design features that enable rapid development of customized routing protocols with wire-speed performance. SwitchBlade has a pipelined architecture (§4.1) with various processing stages. SwitchBlade implements Virtual Data Planes (VDP) (§4.2) so that multiple data plane implementations can be supported on the same platform with performance isolation between the different forwarding protocols. SwitchBlade provides customizable hardware modules (§4.3) that can be enabled or disabled to customize packet processing at runtime. SwitchBlade implements a flexible matching forwarding engine (§4.4) that provides a longest prefix match and an exact hash-based lookup on

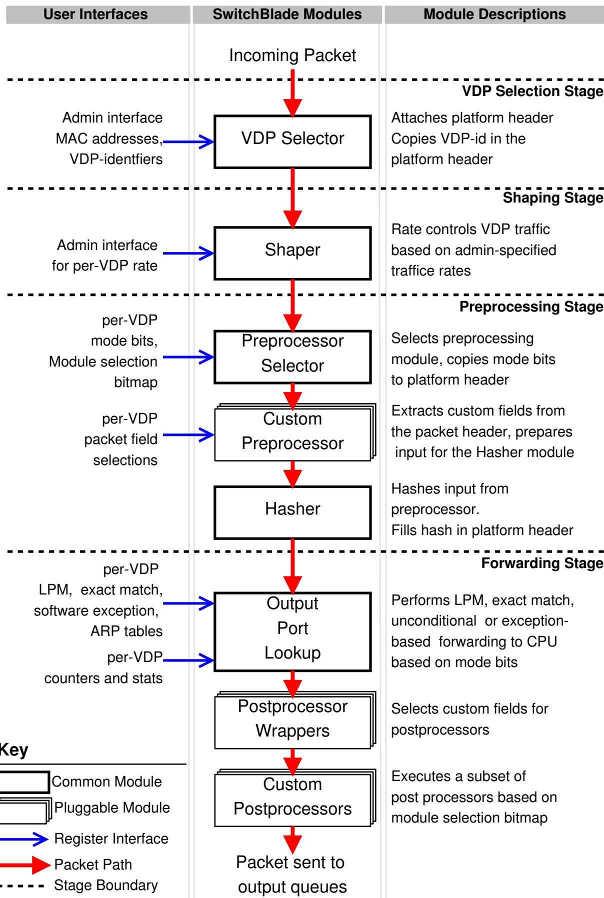

Figure 1: SwitchBlade Packet Processing Pipeline.

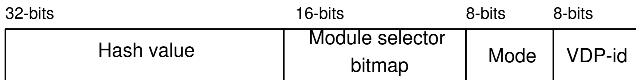

Figure 2: Platform header format. This 64 bit header is applied to every incoming packet and removed before the packet is forwarded.

various fields in the packet header. There are also programmable software exceptions (§4.5) that can be configured from software to direct individual packets or flows to the CPU for additional processing.

4.1 SwitchBlade Pipeline

Figure 1 shows the SwitchBlade pipeline. There are four main stages in the pipeline. Each stage consists of one or more hardware modules. We use a pipelined architecture because it is the most straightforward choice in hardware-based architectures. Additionally, SwitchBlade is based on reference router from the NetFPGA group at Stanford [2]; this reference router has a pipelined architecture as well.

VDP Selection Stage. An incoming packet to SwitchBlade is associated with one of the VDPs. The *VDP Selector* module classifies the packet based on its MAC address and uses a stored table that maps MAC addresses to *VDP identifiers*. A register interface populates the table with the VDP identifiers and is described later.

Field	Value	Description/Action
Mode	0	Default, Perform LPM on IPv4 destination address
	1	Perform exact matching on hash value
	2	Send packet to software for custom processing
	3	Lookup hash in software exceptions table
Module Selector Bitmap	1	Source MAC not updated
	2	Don't decrement TTL
	4	Don't Calculate Checksum
	8	Dest. MAC not updated
	16	Update IPv6 Hop Limit
	32	Use Custom Module 1
	64	Use Custom Module 2
	128	Use Custom Module 3

Table 2: Platform Header: The Mode field selects the forwarding mechanism employed by the Output Port Lookup module. The Module Selector Bitmap selects the appropriate postprocessing modules.

This stage also attaches a 64-bit *platform header* on the incoming packet, as shown in Figure 2. The registers corresponding to each VDP are used to fill the various fields in the platform header. SwitchBlade is a pipelined architecture, so we use a specific header format that to make the architecture extensible. The first byte of this header is used to select the VDP for every incoming packet. Table 2 describes the functionality of the different fields in the platform header.

Shaping Stage. After a packet is designated to a particular VDP, the packet is sent to the shaper module. The shaper module rate limits traffic on per VDP basis. There is a register interface for the module that specifies the traffic rate limits for each VDP.

Preprocessing Stage. This stage includes all the VDP-specific preprocessing hardware modules. Each VDP can customize which preprocessor module in this stage to use for preprocessing the packet via a register interface . In addition to selecting the preprocessor, a VDP can select the various bit fields from the preprocessor using a register interface. A register interface provides information about the mode bits and the preprocessing module configurations. In addition to the custom preprocessing of the packet, this stage also has the hasher module, which can compute a hash of an arbitrary set of bits in the packet header and insert the value of the hash in the packet's platform header.

Forwarding Stage. This final stage in the pipeline handles the operations related to the actual packet forwarding. The *Output Port Lookup* module determines the destination of the packet, which could be one of: (1) longest-prefix match on the packet's destination address field to determine the output port; (2) exact matching on the hash value in the packet's platform header to determine the output port; or (3) exception-based forwarding to the CPU for further processing. This stage uses the mode bits specified in the preprocessing stage. The *Postprocessor Wrappers* and the *Custom Postprocessors* perform operations such as decrementing the packet's time-to-live field. After this stage, SwitchBlade queues the packet in the appropriate output queue for forwarding. SwitchBlade selects the postprocessing module or modules based on the *module selection bits* in the packet's platform header.

4.2 Custom Virtual Data Plane (VDP)

SwitchBlade enables multiple customized data planes top operate simultaneously in parallel on the same hardware. We refer to each data plane as Virtual Data Plane (VDP). SwitchBlade provides a separate packet processing pipeline, as well as separate lookup ta-

bles and register interfaces for each VDP. Each VDP may provide custom modules or share modules with other VDPs. With SwitchBlade, shared modules are not replicated on the hardware, saving valuable resources. Software exceptions include VDP identifiers, making it easy to use separate software handlers for each VDP.

Traffic Shaping. The performance of a VDP should not be affected by the presence of other VDPs. The *shaper* module enables SwitchBlade to limit bandwidth utilization of different VDPs. When several VDPs are sharing the platform, they can send traffic through any of the four ports of the VDP to be sent out from any of the four router ports. Since a VDP can start sending more traffic than what is its bandwidth limit thus affecting the performance of other VDPs. In our implementation, the shaper module comes after the *Preprocessing* stage not before it as shown in Figure 1. This implementation choice, although convenient, does not affect our results because the FPGA data plane can process packets faster than any of the inputs. Hence, the traffic shaping does not really matter. We expect, however, that in the future FPGAs there might be much more than the current four network interfaces for a single NetFPGA which would make traffic shaping of individual VDPs necessary. In the existing implementation, packets arriving at a rate greater than the allocated limit for a VDP are dropped immediately. We made this decision to save memory resources on the FPGA and to prevent any VDP from abusing resources.

Register interface. SwitchBlade provides a register interface for a VDP to control the selection of preprocessing modules, to customize packet processing modules (*e.g.*, which fields to use for calculating hash), and to set rate limits in the shaper module. Some of the values in the registers are accessible by each VDP, while others are only available for the SwitchBlade administrator. SwitchBlade divides the register interfaces into these two security modes: the *admin* mode and the *VDP* mode. The admin mode allows setting of global policies such as traffic shaping, while the VDP mode is for per-VDP module customization.

SwitchBlade modules also provide statistics, which are recorded in the registers and are accessible via the admin interface. The statistics are specific to each module; for example, the VDP selector module can provide statistics on packets accepted or dropped. The admin mode provides access to all registers on the SwitchBlade platform, whereas the VDPmode is only to registers related to a single VDP.

4.3 Customizable Hardware Modules

Rapidly deploying new routing protocols may require custom packet processing. Implementing each routing protocol from scratch can significantly increase development time. There is a significant implementation cycle for implementing hardware modules; this cycle includes design, coding, regression tests, and finally synthesis of the module on hardware. Fortunately, many basic operations are common among different forwarding mechanisms, such as extracting the destination address for lookup, checksum calculation, and TTL decrement. This commonality presents an opportunity for a design that can reuse and even allow sharing the implementations of basic operations which can significantly shorten development cycles and also save precious resources on the FPGA.

SwitchBlade achieves this reuse by providing modules that support a few basic packet processing operations that are common across many common forwarding mechanism. Because SwitchBlade provides these modules as part of its base implementation, data plane protocols that can be composed from only the base modules can be implemented without resynthesizing the hardware and can be programmed purely using a register interface. As an example, to implement a new routing protocol such as Path Splicing [17], which requires manipulation of splicing bits (a custom field in the packet header), a VDP can provide a new module that is included at synthesis time. This module can append preprocessing headers that are later used by SwitchBlade's forwarding engine. A protocol such as OpenFlow [19] may depend only on modules that are already synthesized on the SwitchBlade platform, so it can choose the subset of modules that it needs.

SwitchBlade's reusable modules enable new protocol developers to focus more on the protocol implementation. The developer needs to focus only on bit extraction for custom forwarding. Each pluggable module must still follow the overall timing constraints, but for development and verification purposes, the protocol developer's job is reduced to the module's implementation. Adding new modules or algorithms that offer new functionality of course requires conventional hardware development and must still strictly follow the platform's overall timing constraints.

A challenge with reusing modules is that different VDPs may need the same postprocessing module (*e.g.*, decrementing TTL), but the postprocessing module may need to operate on different locations in the packet header for different protocols. In a naïve implementation, SwitchBlade would have to implement two separate modules, each looking up the corresponding bits in the packet header. This approach doubles the implementation effort and also wastes resources on the FPGA. To address this challenge, SwitchBlade allows a developer to include *wrapper modules* that can customize the behavior of existing modules, within same data word and for same length of data to be operated upon.

As shown in Figure 1 custom modules can be used in the preprocessing and forwarding stages. In the preprocessing stage, the customized modules can be selected by a VDP by specifying the appropriate selection using the register interface. Figure 3 shows an example: the incoming packet from the previous shaping stage which goes to a demultiplexer which selects the appropriate module or modules for the packet based on the input from the register interface specific to the particular VDP that the packet belongs to. After being processed by one of the protocol modules (*e.g.*, IPv6, OpenFlow), the packet arrives at the hasher module. The hasher module takes 256 bits as input and generates a 32-bit hash of the input. The hasher module need not be restricted to 256 bits of input data, but a larger input data bus would mean using more resources. Therefore, we decided to implement a 256-bit wide hash data bus to accommodate our design on the NetFPGA.

Each VDP can also use custom modules in the forwarding stage, by selecting the appropriate postprocessor wrappers and custom postprocessor modules as shown in Figure 1. SwitchBlade selects these modules based on the *module selection bitmap* in the platform header of the packet. Figure 4(b) shows an example of the custom wrapper and postprocessor module selection operation.

4.4 Flexible Matching for Forwarding

New routing protocols often require customized routing tables, or forwarding decisions on customized fields in the packet. For example, Path Splicing requires multiple IP-based forwarding tables, and the router chooses one of them based on splicing bits in the packet header. SEATTLE [15] and Portland [18] use MAC address-based forwarding. Some of the forwarding mechanisms are still simple enough to be implemented in hardware and can benefit from fast-path forwarding; others might be more complicated and it might be easier to just have the forwarding decision be made in software. Ideally, all forwarding should take place in hardware, but there is a tradeoff in terms of forwarding performance and hardware implementation complexity.

Preprocessor Selection		
Code	Processor	Description
1	Custom Extractor	Allows selection of variable 64-bit fields in packet on 64-bit boundaries in first 32 bytes
2	OpenFlow	OpenFlow packet processor that allows variable field selection.
3	Path Splicing	Extracts Destination IP Address and uses bits in packet to select the Path/Forwarding Table.
4	IPv6	Extracts IPv6 destination address.

Table 3: Processor Selection Codes.

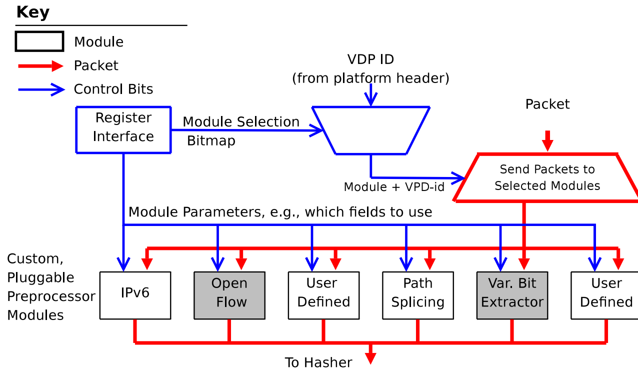

Figure 3: Virtualized, Pluggable Module for Programmable Processors.

SwitchBlade uses a hybrid hardware-software approach to strike a balance between forwarding performance and implementation complexity. Specifically, SwitchBlade's forwarding mechanism implementation, provided by the *Output Port Lookup* module as shown in Figure 1, provides the following four different methods for making forwarding decision on the packet: (1) conventional longest prefix matching (LPM) on any 32-bit address field in the packet header within the first 40-bytes; (2) exact matching on hash value stored in the packet's platform header; (3) unconditionally sending the packet to the CPU for making the forwarding computation; and (4) sending only packets which match certain user defined exceptions, called software exceptions 4.5, to the CPU. The details of how the output port lookup module performs these tasks is illustrated in Figure 4(a). Modes (1) and (2) enable fast-path packet forwarding because the packet never leaves the FPGA. We observe that many common routing protocols can be implemented with these two forwarding mechanisms alone. Figure 4 is not the actual implementation but shows the functional aspect of SwitchBlade's implementation.

By default, SwitchBlade performs a longest-prefix match, assuming an IPv4 destination address is present in the packet header. To enable use of customized lookup, a VDP can set the appropriate mode bit in the *platform header* of the incoming packet. One of the four different forwarding mechanisms can be invoked for the packet by the mode bits as described in Table 2. The output port lookup module performs LPM and exact matching on the hash value from the forwarding table stored in the TCAM. The same TCAM is used for LPM and for exact matching for hashing therefore the mask from the user decides the nature of match being done.

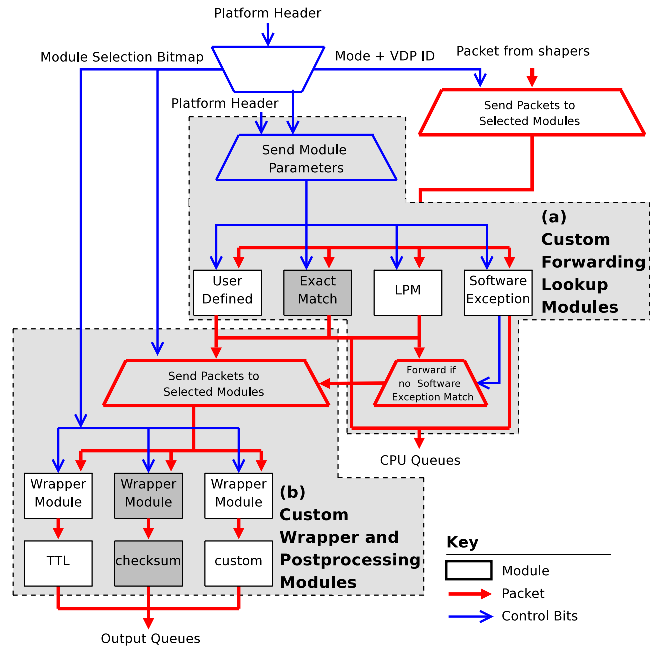

Figure 4: Output Port Lookup and Postprocessing Modules.

Once the output port lookup module determines the output port for the packet it adds the output port number to the packet's platform header. The packet is then sent to the postprocessing modules for further processing. In Section 4.5, we describe the details of software work and how the packet is handled when it is sent to the CPU.

4.5 Flexible Software Exceptions

Although performing all processing of the packets in hardware is the only way to achieve line rate performance, it may be expensive to introduce complex forwarding implementations in the hardware. Also, if certain processing will only be performed on a few packets and the processing requirements of those packets are different from the majority of other packets, development can be faster and less expensive if those few packets are processed by the CPU instead (*e.g.*, ICMP packets in routers are typically processed in the CPU).

SwitchBlade introduces *software exceptions* to programmatically direct certain packets to the CPU for additional processing. This concept is similar to the OpenFlow concept of rules that can identify packets that match a particular traffic flow that should be passed to the controller. However, combining software exceptions with the LPM table provides greater flexibility, since a VDP can add exceptions to existing forwarding rules. Similarly, if a user starts receiving more traffic than expected from a particular software exception, that user can simply remove the software exception entry and add the forwarding rule in forwarding tables.

There is a separate exceptions table, which can be filled via a register interface on a per-VDP basis and is accessible to the output port lookup module, as shown in Figure 4(a). When the mode bits field in the platform header is set to 3 (Table 2), the output port lookup module performs an exact match of the hash value in the packet's platform header with the entries in the exceptions table for the VDP. If there is a match, then the packet is redirected to the CPU where it can be processed using software-based handlers, and if there is none then the packet is sent back to the output port

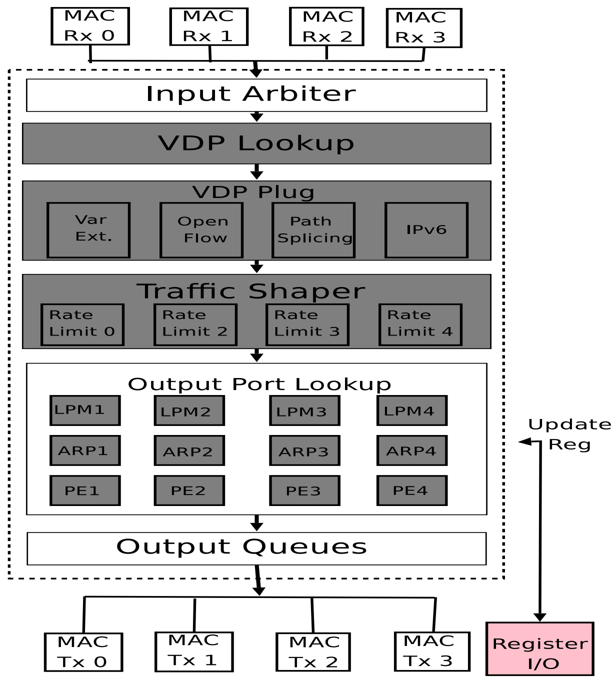

Figure 5: SwitchBlade Pipeline for NetFPGA implementation.

lookup module to perform an LPM on the destination address. We describe the process after the packet is sent to the CPU later.

SwitchBlade's software exceptions feature allows decision caching [9]: software may install its decisions as LPM or exact match rules in the forwarding tables so that future packets are forwarded rapidly in hardware without causing software exceptions.

SwitchBlade allows custom processing of some packets in software. There are two forwarding modes that permit this function: unconditional forwarding of all packets or forwarding of packets based on software exceptions to the CPU. Once a packet has been designated to be sent to the CPU, it is placed in a CPU queue corresponding to its VDP, as shown in Figure 4(a). The current SwitchBlade implementation forwards the packet to the CPU, with the platform header attached to the packet. We describe one possible implementation of a software component on top of SwitchBlade's VDP—a virtual router—in Section 7.

5. NETFPGA IMPLEMENTATION

In this section, we describe our NetFPGA-based implementation of SwitchBlade, as well as custom data planes that we have implemented using SwitchBlade. For each of these data planes, we present details of the custom modules, and how these modules are integrated into the SwitchBlade pipeline.

5.1 SwitchBlade Platform

SwitchBlade implements all the modules shown in Figure 5 on the NetFPGA [2] platform. The current implementation uses four packet preprocessor modules, as shown in Table 3. SwitchBlade uses SRAM for packet storage and BRAM and SRL16e storage for forwarding information for all the VDPs 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 to add other features, such as the support of virtual data planes, customizable hardware modules, and programmable software exceptions. Figure 5 shows the implementation of the NetFPGA router-based pipeline for SwitchBlade. Because our implementation is based on the NetFPGA reference implementation, adding multicast packet forwarding depends on the capabilities of NetFPGA reference router [2] implementation. Because the base implementation can support multicast forwarding, SwitchBlade can also support it.

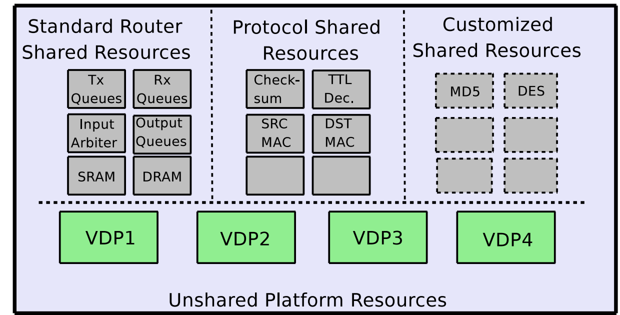

Figure 6: Resource sharing in SwitchBlade.

VDP Selection Stage. The SwitchBlade implementation adds three new stages to the NetFPGA reference router [2] pipeline as shown in gray in Figure 5. The VDP selection stage essentially performs destination MAC lookup for each incoming packet and if the destination MAC address matches then the packet is accepted and the VDP-id is attached to the packet's platform header (Table 2). VDP selection is implemented using a CAM (Content Addressable Memory), where each MAC address is associated with a VDP-ID. This table is called the *Virtual Data Plane table*. An admin register interface allows the SwitchBlade administrator to allow or disallow users from using a VDP by adding or removing their destination MAC entries from the table.

Preprocessing Stage. A developer can add customizable packet preprocessor modules to the VDP. There are two main benefits for these customizable preprocessor modules. First, this modularity streamlines the deployment of new forwarding schemes. Second, the hardware cost of supporting new protocols does not increase linearly with the addition of new protocol preprocessors. To enable custom packet forwarding, the preprocessing stage also provides a hashing module that takes 256-bits as input and produces a 32-bit output (Table 2). The hashing scheme does not provide a longest-prefix match; it only offers support for an exact match on the hash value. In our existing implementation each preprocessor module is fixed with one specific VDP.

Shaping Stage. We implement bandwidth isolation for each VDP using a simple network traffic rate limiter. Each VDP has a configurable rate limiter that increases or decreases the VDP's allocated bandwidth. We used a rate limiter from the NetFPGA's reference implementation for this purpose. The register interface to update the rate limits is accessible only with admin privileges.

Software Exceptions. To enable programmable software exceptions, SwitchBlade uses a 32-entry CAM within each VDP that can be configured from software using the register interface. SwitchBlade has a register interface that can be used to add a 32-bit hash representing a flow or packet. Each VDP has a set of registers to update the software exceptions table to redirect packets from hardware to software.

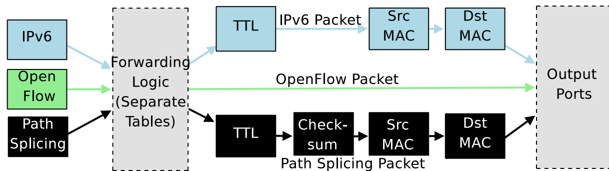

Figure 7: Life of OpenFlow, IPv6, and Path Splicing packets.

Sharing and custom packet processing. The modules that function on the virtual router instance are shared between different virtual router instances that reside on the same FPGA device. Only those modules that the virtual router user selects can operate on the packet; others do not touch the packet. This path-selection mechanism is unique. Depending on an individual virtual router user's requirements, the user can simply select the path of the packet and the modules that the virtual router user requires.

5.2 Custom Data Planes using SwitchBlade

Implementing any new functionality in SwitchBlade requires hardware programming in Verilog, but if the module is added as a pluggable preprocessor, then the developer needs to be concerned with the pluggable preprocessor implementation only, as long as decoding can occur within specific clock cycles. Once a new module is added and its interface is linked with the register interface, a user can write a high-level program to use a combination of the newly added and previously added modules. Although the number of modules in a pipeline may appear limited because of smaller header size, this number can be increased by making the pipeline wider or by adding another header for every packet.

To allow developers to write their own protocols or use existing ones, SwitchBlade offers header files in C++, Perl, and Python; these files refer to register address space for that user's register interface only. A developer simply needs to include one of these header files. Once the register file is included, the developer can write a user-space program by reading and writing to the register interface. The developer can then use the register interface to enable or disable modules in the SwitchBlade pipeline. The developer can also use this interface to add hooks for software exceptions. Figure 7 shows SwitchBlade's custom packet path. We have implemented three different routing protocols and forwarding mechanisms: OpenFlow [19], Path Splicing [17], and IPv6 [11] on SwitchBlade.

OpenFlow. We implemented the exact match lookup mechanism of OpenFlow in hardware using SwitchBlade without VLAN support. The OpenFlow preprocessor module, as shown in Figure 3, parses a packet and extracts the ten tuples of a packet defined in OpenFlow specifications. The OpenFlow preprocessor module extracts the bits from the packet header and returns a 240-bit wide OpenFlow flow entry. These 240-bits travel on a 256-bit wire to the hasher module. The hasher module returns a 32-bit hash value that is added to the SwitchBlade platform header (Figure 2). After the addition of hash value this module adds a module selector bitmap to the packet's platform header. The pipeline then sets mode field in the packet's platform header to 1, which makes the output port lookup module perform an exact match on the hash value of the packet. The output port lookup module looks up the hash value in the exact match table and forwards the packet to the output port if the lookup was a hit. If the table does not contain the corresponding entry, the platform forward the packet to the CPU for processing with software-based handlers.

Because OpenFlow offers switch functionality and does not require any extra postprocessing (*e.g.*, TTL decrement or checksum calculation), a user can prevent the forwarding stage from performing any extra postprocessing functions on the packet. Nothing happens in the forwarding stage apart from the lookup, and SwitchBlade queues the packet in the appropriate output queue. A developer can update source and destination MACs as well, using the register interface.

Path Splicing. Path Splicing enables users to select different paths to a destination based on the *splicing bits* in the packet header. The splicing bits are included as a bitmap in the packet's header and serve as an index for one of the possible paths to the destination. To implement Path Splicing in hardware, we implemented a processing module in the preprocessing stage. For each incoming packet, the preprocessor module extracts the splicing bits and the destination IP address. It concatenates the IP destination address and the splicing bits to generate a new address that represents a separate path. Since Path Splicing allows variation in path selection, this bit field can vary in length. The hasher module takes this bit field, creates a 32-bit hash value, and attaches it to the packet's platform header.

When the packet reaches the exact match lookup table, its 32-bit hash value is extracted from SwitchBlade header and is looked up in the exact match table. If a match exists, the card forwards the packet on the appropriate output port. Because the module is concatenating the bits and then hashing them and there is an exact match down the pipeline, two packets with the same destination address but different paths will have different hashes, so they will be matched against different forwarding table entries and routed along two different paths. Since Path Splicing uses IPv4 for packet processing, all the postprocessing modules on the default path (*e.g.*, TTL decrement) operate on the packet and update the packet's required fields. SwitchBlade can also support equal-cost multipath (ECMP). For this protocol, the user must implement a new preprocessor module that can select two different paths based on the packet header fields and can store their hashes in the lookup table sending packets to two separate paths based on the hash match in lookup.

IPv6. The IPv6 implementation on SwitchBlade also uses the customizable preprocessor modules to extract the 128-bit destination address from an incoming IPv6 packet. The preprocessor module extracts the 128-bits and sends them to the hasher module to generate a 32-bit hash from the address.

Our implementation restricts longest prefix match to 32-bit address fields, so it is not currently possible to perform longest prefix match for IPv6. The output port lookup stage performs an exact match on the hash value of the IPv6 packet and sends it for postprocessing. When the packet reaches the postprocessing stage, it only needs to have its TTL decremented because there is no checksum in IPv6. But it also requires to have its source and destination MACs updated before forwarding. The module selector bitmap shown in Figure 5 enables only the postprocessing module responsible for TTL decrement and not the ones doing checksum recalculation. Because the TTL offset for IPv6 is at a different byte offset than the default IPv4 TTL field, SwitchBlade uses a wrapper module that extracts only the bits of the packet's header that are required by the TTL decrement module; it then updates the packet's header with the decremented TTL.

Resource	NetFPGA Utilization	% Utilization
Slices	21 K out of 23 K	90%
4-input LUTs	37 K out of 47 K	79%
Flip Flops	20 K out of 47 K	42%
External IOBs	353 out of 692	51%
Eq. Gate Count	13 M	N/A

Table 4: Resource utilization for the base SwitchBlade platform.

6. EVALUATION

In this section, we evaluate our implementation of SwitchBlade using NetFPGA [2] as a prototype development platform. Our evaluation focuses on three main aspects of SwitchBlade: (1) resource utilization for the SwitchBlade platform; (2) forwarding performance and isolation for parallel data planes; and (3) data-plane update rates.

6.1 Resource Utilization

To provide insight about the resource usage when different data planes are implemented on SwitchBlade, we used Xilinx ISE [23] 9.2 to synthesize SwitchBlade. We found that a single physical IPv4 router implementation developed by the NetFPGA group at Stanford University uses a total of $23K$ four-input LUTs, which consume about 49% of the total available four-input LUTs, on the NetFPGA. The implementation also requires 123 BRAM units, which is 53% of the total available BRAM.

We refer to our existing implementation with one OpenFlow, one IPv6, one variable bit extractor, and one Path Splicing preprocessor with an IPv4 router and capable of supporting four VDPs as the SwitchBlade"base configuration". This implementation uses $37K$ four-input LUTs, which account for approximately 79% of four-input LUTs. Approximately 4.5% of LUTs are used for shift registers. Table 4 shows the resource utilization for the base SwitchBlade implementation; SwitchBlade uses more resources than the base IPv4 router, as shown in table 5, but the increase in resource utilization is less than linear in the number of VDPs that SwitchBlade can support.

Sharing modules enables resource savings for different protocol implementations. Table 5 shows the resource usage for implementations of an IPv4 router, an OpenFlow switch, and path splicing. These implementations achieve 4 Gbps; OpenFlow and Path Splicing implementations provide more resources than SwitchBlade. But there is not much difference in resource usage for these implementations when compared with the possible configurations which SwitchBlade can support.

Virtual Data Planes can support multiple forwarding planes in parallel. Placing four Path Splicing implementations in parallel on a larger FPGA to run four Path Splicing data planes will require four times the resources of existing Path Splicing implementation. Because no modules are shared between the four forwarding planes, the number of resources will not increase linearly and will remain constant in the best case.

From a gate count perspective, Path Splicing with larger forwarding tables and more memory will require approximately four times the resources as in Table 5; SwitchBlade with smaller forwarding tables and less memory will require almost same amount of resources. This resource usage gap begins to increase as we increase the number of Virtual Data Planes on the FPGA. Recent trends in FPGA development such as Virtex 6 suggest higher speeds and larger area; these trends will allow more VDPs to be placed on a single FPGA, which will facilitate more resource sharing.

NetFPGA Implementation	Slices	4-input LUTs	Flip Flops	BRAM	Equivalent Gate Count
Path Splicing	17 K	19 K	17 K	172	12 M
OpenFlow	21 K	35 K	22 K	169	12 M
IPv4	16 K	23 K	15 K	123	8 M

Table 5: Resource usage for different data planes.

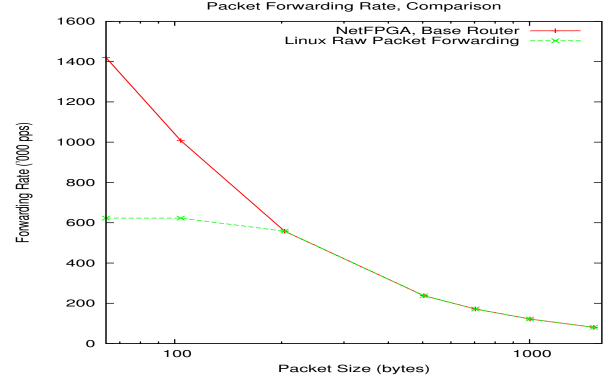

Figure 8: Comparison of forwarding rates.

6.2 Forwarding Performance and Isolation

We used the NetFPGA-based packet generator [10] for traffic generation to generate high speed traffic to evaluate the forwarding performance of SwitchBlade and the isolation provided between the VDPs. Some of the results we present in this section are derived from experiments in previous work [4].

Raw forwarding rates. Previous work has measured the maximum sustainable packet forwarding rate for different configurations of software-based virtual routers [5]. We also measure packet forwarding rates and show that hardware-accelerated forwarding can increase packet forwarding rates. We compare forwarding rates of Linux and NetFPGA-based router implementation from NetFPGA group [2], as shown in Figure 8. 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 see packet drops in hardware. If we impose the condition that no packets should be dropped at the router, then the packet forwarding rates for the Linux router drops significantly, but the forwarding rates for the hardware-based router remain constant. Figure 8 shows packet forwarding rates when this "no packet drop" condition is *not* imposed (*i.e.*, we measure the maximum sustainable forwarding rates). For large packet sizes, SwitchBlade could achieve the same forwarding rate using in-kernel forwarding as we were using a single port of NetFPGA router. Once the packet size drops below 200 bytes; the software-based router cannot keep pace with the forwarding requirements.

Forwarding performance for Virtual Data Planes. Figure 9 shows the data-plane forwarding performance of SwitchBlade running four data planes in parallel versus the NetFPGA reference router [2], for various packet sizes. We have disabled the rate limiters in SwitchBlade for these experiments. The figure shows that running SwitchBlade incurs no additional performance penalty when compared to the performance of running the refer-

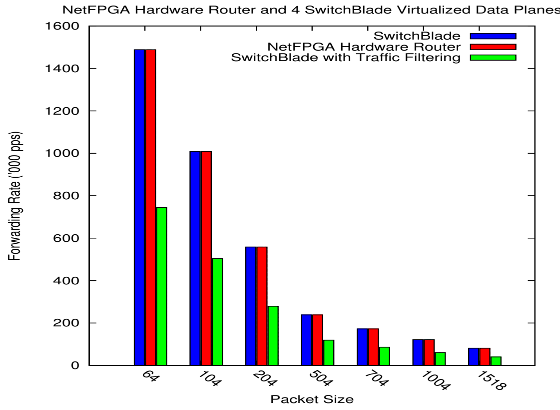

Figure 9: Data plane performance: NetFPGA reference router vs. SwitchBlade.

Physical Router ('000s of packets)			
Packet Size(bytes)	Pkts Sent	Pkts Fwd @ Core	Pkts recv @Sink
64	40 K	40 K	20 K
104	40 K	40 K	20 K
204	40 K	40 K	20 K
504	40 K	40 K	20 K
704	40 K	40 K	20 K
1004	39.8 K	39.8 K	19.9 K
1518	4 K	4 K	1.9 K

Table 6: Physical Router, Packet Drop Behavior.

ence router [2]. By default, traffic belonging to any VDP can arrive on any of the physical Ethernet interfaces since all of the ports are in promiscuous mode. To measure SwitchBlade's to filter traffic that is not destined for any VDP, we flooded SwitchBlade with a mix of traffic where half of the packets had destination MAC addresses of SwitchBlade virtual interfaces and half of the packets had destination MAC addresses that didn't belong to any vdp. As a result, half of the packets were dropped and rest were forwarded, which resulted in a forwarding rate that was half of the incoming traffic rate.

Isolation for Virtual Data Planes. To measure CPU isolation, we used four parallel data planes to forward traffic when a user-space process used 100% of the CPU. We then sent traffic where each user had an assigned traffic quota in packets per second. When no user surpassed the assigned quotas, the router forwarded traffic according to the assigned rates, with no packet loss. To measure traffic isolation, we set up a topology where two 1 Gbps ports of routers were flooded at 1 Gbps and a sink node were connected to a third 1 Gbps port. We used four VDPs to forward traffic to the same output port. Tables 6 and 7 show that, at this packet forwarding rate, only half of the packets make it through, on first come first serve basis, as shown in fourth column. These tables show that both the reference router implementation and SwitchBlade have the same performance in the worst-case scenario when an output port is flooded. The second and third columns show the number of packets sent to the router and the number of packets forwarded by the router. Our design does not prevent against contention that may arise when many users send traffic to one output port.

Forwarding performance for non-IP packets. We also tested whether SwitchBlade incurred any forwarding penalty for forwarding custom, non-IP packets; SwitchBlade was also able to forward these packets at the same rate as regular IP packets. Figure 10

Four Data Planes ('000s of packets)			
Packet Size(bytes)	Pkts Sent	Pkts Fwd @ Core	Pkts recv @Sink
64	40 K	40 K	20 K
104	40 K	40 K	20 K
204	40 K	40 K	20 K
504	40 K	40 K	20 K
704	40 K	40 K	20 K
1004	39.8 K	39.8 K	19.9 K
1518	9.6 K	9.6 K	4.8 K

Table 7: Four Parallel Data Planes, Packet Drop Behavior.

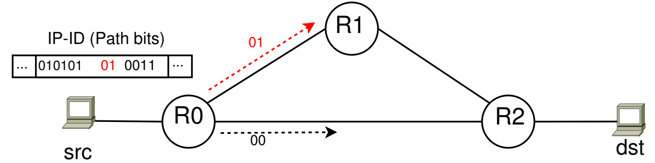

Figure 10: Test topology for testing SwitchBlade implementation of Path Splicing.

shows the testbed we used to test the Path Splicing implementation. We again used the NetFPGA-based hardware packet generator [10] to send and receive traffic. Figure 11 shows the packet forwarding rates of this NetFPGA-based implementation, as observed at the sink node. No packet loss occurred on any of the nodes shown in Figure 10. We sent two flows with same destination IP address but using different splicing bits to direct them to different routers. Packets from one flow were sent to $R2$ via $R1$, while others went directly to $R2$. In another iteration, we introduced four different flows in the network, such that all four forwarding tables at router $R0$ and $R2$ were looked up with equal probability; in this case, SwitchBlade also forwarded the packets at full rate. Both these experiments show that SwitchBlade can implement schemes like Path Splicing and forward traffic at hardware speeds for non-IP packets.

In another experiment, we used the Variable Bit Extraction module to extract first 64 bits from the header for hashing. We used a simple source and sink topology with SwitchBlade between them and measured the number of packets forwarded. Figure 12 shows the comparison of forwarding rates when forwarding was being done using SwitchBlade based on the first 64-bits of an Ethernet frame and when it was done using NetFPGA base router.

6.3 Data-Plane Update Rates

Each VDP in a router on SwitchBlade needs to have its own forwarding table. Because the VDPs share a single physical device, simultaneous table updates from different VDPs might create a bottleneck. To evaluate the performance of SwitchBlade for forwarding table update speeds, we assumed the worst-case scenario, where all VDPs flush their tables and rewrite them again at the same time. We assumed that the table size for each VDP is 400,000 entries. We updated all four tables simultaneously, but there was no performance decrease while updating the forwarding table from software. Four processes were writing the table entries in the forwarding table.

Table 8 shows updating 1.6 million entries simultaneously took 89.77 seconds on average, with a standard deviation of less than one second. As the number of VDPs increases, the average update rate remains constant, but as the number of VDPs increases, the PCI interconnect speed becomes a bottleneck between the VDP processes updating the table and the SwitchBlade FPGA.

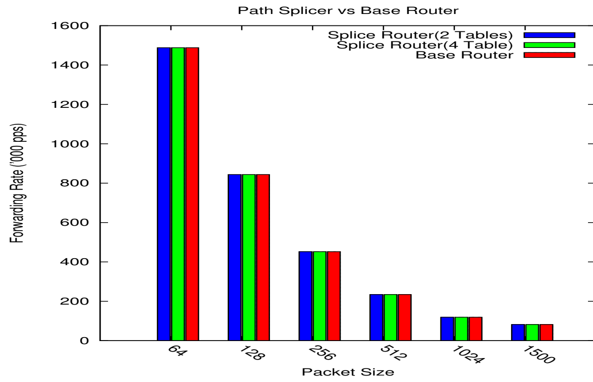

Figure 11: Path Splicing router performance with varying load compared with base router.

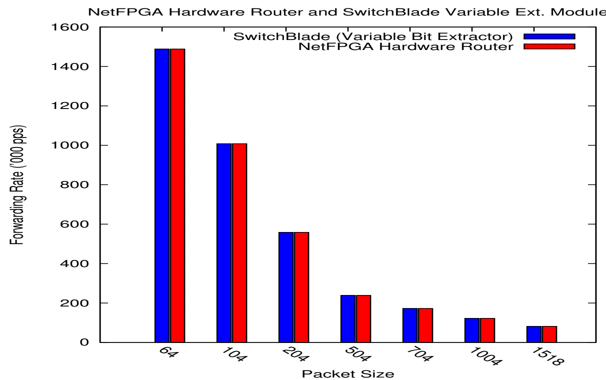

Figure 12: Variable Bit Length Extraction router performance compared with base router.

7. A VIRTUAL ROUTER ON SWITCHBLADE

We now describe our integration of SwitchBlade with a virtual router environment that runs in an OpenVZ container [20]. We use OpenVZ [20] as the virtual environment for hosting virtual routers for two reasons related to isolation. First, OpenVZ provides some level of namespace isolation between each of the respective virtual environments. Second, OpenVZ provides a CPU scheduler that prevents any of the control-plane processes from using more than its share of CPU or memory resources. We run the Quagga routing software [21] in OpenVZ, as shown in Figure 13.

Each virtual environment has a corresponding VDP that acts as its data plane. SwitchBlade exposes a register interface to send commands from the virtual environment to its respective VDP. Similarly, each VDP can pass data packets into its respective virtual environment using the software exception handling mechanisms described in Section 4.5.

We run four virtual environments on the same physical machine and use the SwitchBlade's isolation capabilities to share the hardware resources. Each virtual router receives a dedicated amount of processing and is isolated from the other routers' virtual data planes. Each virtual router also has the appearance of a dedicated data path.

8. DISCUSSION

PCIe interconnect speeds and the tension between hardware and software. Recent architectures for software-based routers such as RouteBricks, use the PCI express (PCIe) interface between the

VDPs	Total Ent.	Entries/Table	Time(sec)	Single Ent. (μs)
1	400 K	400 K	86.582	216
2	800 K	400 K	86.932	112
3	1,200 K	400 K	88.523	74
4	1,600 K	400 K	89.770	56

Table 8: Forwarding Table Update Performance.

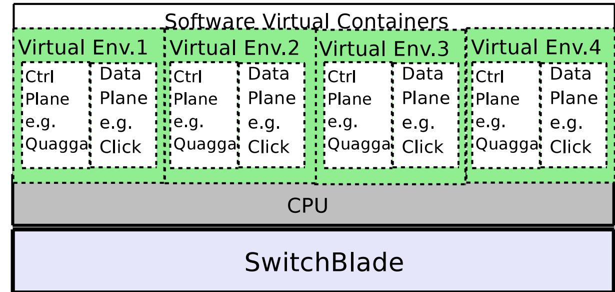

Figure 13: Virtual router design with OpenVZ virtual environments interfacing to SwitchBlade data plane.

CPU, which acts as the I/O hub, and the network interface cards that forward traffic. PCIe offers more bandwidth than a standard PCI interface; for example, PCIe version 2, with 16 lanes link, has a total aggregate bandwidth of 8 GBps per direction. Although this high PCIe bandwith would seem to offer great promise for building programmable routers that rely on the CPU for packet processing, the speeds of programmable interface cards are also increasing, and it is unclear as yet whether the trends will play out in favor of CPU-based packet processing. For example, one Virtex-6 HXT FPGA from Xilinx or Stratix V FPGA from Altera can process packets at 100 Gbps. Thus, installing NICs with only one such FPGA can make the PCIe interconnect bandwidth a bottleneck, and also puts an inordinate amount of strain on the CPU. SwitchBlade thus favors making FPGAs more flexible and programmable, allowing more customizability to take place directly on the hardware itself.

Modifying packets in hardware. SwitchBlade's hardware implementation focuses on providing customization for protocols that make only limited modifications to packets. The design can accommodate writing packets using preprocessor modules, but we have not yet implemented this function. Providing arbitrary writing capability in hardware will require either using preprocessor stage for packet writing and a new pipeline stage after postprocessing, or adding two new stages to the pipeline (both before and after lookup).

Packet rewriting can be performed in two ways: (1) modifying existing packet content without changing total data unit size, or (2) adding or removing some data to each packet with the output packet size different from the input packet size. Although it is easy to add the first function to the preprocessor stage, adding or removing bytes into packet content will require significant effort.

Scaling SwitchBlade. The current SwitchBlade implementation provides the capability for four virtualized data planes on a single NetFPGA, but this design is general enough to scale as the capabilities of hardware improve. We see two possible avenues for increasing the number of virtualized data planes in hardware. One option is to add several servers, each having one FPGA card and have one or more servers running the control plane that controls the hardware forwarding-table entries. Other scaling options include adding more FPGA cards to a single physical machine or

taking advantage of hardware trends, which promise the ability to process data in hardware at increasingly higher rates.

9. CONCLUSION

We have presented the design, implementation, and evaluation of SwitchBlade, a platform for deploying custom protocols on programmable hardware. SwitchBlade uses a pipeline-based hardware design; using this pipeline, developers can swap common hardware processing modules in and out of the packet-processing flow on the fly, without having to resynthesize hardware. SwitchBlade also offers programmable software exception handling to allow developers to integrate custom functions into the packet processing pipeline that cannot be handled in hardware. SwitchBlade's customizable forwarding engine also permits the platform to make packet forwarding decisions on various fields in the packet header, enabling custom, non-IP based forwarding at hardware speeds. Finally, SwitchBlade can host multiple data planes in hardware in parallel, sharing common hardware processing modules while providing performance isolation between the respective data planes. These features make SwitchBlade a suitable platform for hosting virtual routers or for simply deploying multiple data planes for protocols or services that offer complementary functions in a production environment. We implemented SwitchBlade using the NetFPGA platform, but SwitchBlade can be implemented with any FPGA.

Acknowledgments

This work was funded by NSF CAREER Award CNS-0643974 and NSF Award CNS-0626950. We thank Mohammad Omer for his help in solving various technical difficulties during project. We also thank our shepherd, Amin Vahdat, for feedback and comments that helped improve the final draft of this paper.

REFERENCES

[1] FlowVisor. http://www.openflowswitch.org/wk/index.php/FlowVisor.

[2] NetFPGA. http://www.netfpga.org.

[3] 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.

[4] M. B. Anwer and N. Feamster. Building a Fast, Virtualized Data Plane with Programmable Hardware. In *Proc. ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, Barcelona, Spain, Aug. 2009.

[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, Georgia Institute of Technology, Atlanta, GA, Oct. 2007.

[6] S. Bhatia, M. Motiwala, W. Mühlbauer, V. Valancius, A. Bavier, N. Feamster, J. Rexford, and L. Peterson. Hosting virtual networks on commodity hardware. Technical Report GT-CS-07-10, College of Computing, Georgia Tech, Oct. 2007.

[7] G. Calarco, C. Raffaelli, G. Schembra, and G. Tusa. Comparative analysis of smp click scheduling techniques. In *QoS-IP*, pages 379–389, 2005.

[8] L. D. Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam. Flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *Proc. ACM SIGCOMM*, Barcelona, Spain, Aug. 2009.

[9] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking packet forwarding hardware. In *Proc. Seventh ACM SIGCOMM HotNets Workshop*, Nov. 2008.

[10] G. A. Covington, G. Gibb, J. Lockwood, and N. McKeown. A Packet Generator on the NetFPGA platform. In *FCCM '09: IEEE Symposium on Field-Programmable Custom Computing Machines*, 2009.

[11] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. Internet Engineering Task Force, Dec. 1998. RFC 2460.

[12] M. Dobrescu, , N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[13] B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica. Pathlet routing. In *Proc. ACM SIGCOMM*, Barcelona, Spain, Aug. 2009.

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

[15] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A scalable ethernet architecture for large enterprises. In *Proc. ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[17] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path Splicing. In *Proc. ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[18] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer2 data center network fabric. In *Proc. ACM SIGCOMM*, Barcelona, Spain, Aug. 2009.

[19] OpenFlow Switch Consortium. http://www.openflowswitch.org/, 2008.

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

[21] Quagga software routing suite. http://www.quagga.net/.

[22] 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.

[23] Xilinx. Xilinx ise design suite. http://www.xilinx.com/tools/designtools.htm.

[24] X. Yang, D. Wetherall, and T. Anderson. Source selectable path diversity via routing deflections. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.