ServerSwitch: A Programmable and High Performance Platform for Data Center Networks

Guohan Lu, Chuanxiong Guo,[*] Yulong Li, Zhiqiang Zhou[††]
Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, Yongguang Zhang
Microsoft Research Asia, Beijing, China
[†] *Tsinghua University, Beijing, China*

Abstract

As one of the fundamental infrastructures for cloud computing, data center networks (DCN) have recently been studied extensively. We currently use pure software-based systems, FPGA based platforms, *e.g.*, NetFPGA, or OpenFlow switches, to implement and evaluate various DCN designs including topology design, control plane and routing, and congestion control. However, software-based approaches suffer from high CPU overhead and processing latency; FPGA based platforms are difficult to program and incur high cost; and OpenFlow focuses on control plane functions at present.

In this paper, we design a *ServerSwitch* to address the above problems. ServerSwitch is motivated by the observation that commodity Ethernet switching chips are becoming programmable and that the PCI-E interface provides high throughput and low latency between the server CPU and I/O subsystem. ServerSwitch uses a commodity switching chip for various customized packet forwarding, and leverages the server CPU for control and data plane packet processing, due to the low latency and high throughput between the switching chip and server CPU.

We have built our ServerSwitch at low cost. Our experiments demonstrate that ServerSwitch is fully programmable and achieves high performance. Specifically, we have implemented various forwarding schemes including source routing in hardware. Our in-network caching experiment showed high throughput and flexible data processing. Our QCN (Quantized Congestion Notification) implementation further demonstrated that ServerSwitch can react to network congestions in 23us.

1 Introduction

Data centers have been built around the world for various cloud computing services. Servers in data centers are interconnected using data center networks. A large data center network may connect hundreds of thousands of servers. Due to the rise of cloud computing, data center networking (DCN) is becoming an important area of research. Many aspects of DCN, including topology design and routing [15, 5, 13, 11, 22], flow scheduling and congestion control [7, 6], virtualization [14], application support [26, 4], have been studied.

Since DCN is a relatively new exploration area, many of the designs (*e.g.*, [15, 5, 13, 22, 7, 14, 4]) have departed from the traditional Ethernet/IP/TCP based packet format, Internet-based single path routing (*e.g.*, OSPF), and TCP style congestion control. For example, Portland performs longest prefix matching (LPM) on destination MAC address, BCube advocates source routing, and QCN (Quantized Congestion Notification) [7] uses rate-based congestion control. Current Ethernet switches and IP routers therefore cannot be used to implement these designs.

To implement these designs, rich programmability is required. There are approaches that provide this programmability: pure software-based [17, 10, 16] or FPGA-based systems (*e.g.*, NetFPGA [23]). Software-based systems can provide full programmability and as recent progress [10, 16] has shown, may provide a reasonable packet forwarding rate. But their forwarding rate is still not comparable to commodity switching ASICs (application specific integrated circuit), and the batch processing used in their optimizations introduces high latency which is critical for various control plane functions such as signaling and congestion control [13, 22, 7]. Furthermore, the packet forwarding logics in DCN (*e.g.*, [15, 13, 22, 14]) are generally simple and hence are better implemented in silicon for cost and power savings. FPGA-based systems are fully programmable. But

[*] Correspondent author.

[†] This work was performed when Zhiqiang Zhou was a visiting student at Microsoft Research Asia.

the programmability is provided by hardware description languages such as Verilog, which are not as easy to learn and use as higher-level programming languages such as C/C++. Furthermore, FPGAs are expensive and are difficult to use in large volumes in data center environments.

In this paper, we design a ServerSwitch platform, which provides easy-to-use programmability, low latency and high throughput, and low cost. ServerSwitch is based on two observations as follows. First, we observe that commodity switching chips are becoming programmable. Though the programmability is not comparable to general-purpose CPUs, it is powerful enough to implement various packet forwarding schemes with different packet formats. Second, current standard PCI-E interface provides microsecond level latency and tens of Gb/s throughput between the I/O subsystem and server CPU. ServerSwitch is then a commodity server plus a commodity, programmable switching chip. These two components are connected via the PCI-E interface.

We have designed and implemented ServerSwitch. We have built a ServerSwitch card, which uses a merchandise gigabit Broadcom switching chip. The card connects to a commodity server using a PCI-E X4 interface. Each ServerSwitch card costs less than 400$ when manufactured in 100 pieces. We also have implemented a software stack, which manages the card, and provides support for control and data plane packet processing. We evaluated ServerSwitch using micro benchmarks and real DCN designs. We built a ServerSwitch based, 16-server BCube [13] testbed. We compared the performance of software-based packet forwarding and our ServerSwitch based forwarding. The results showed that ServerSwitch achieves high performance and zero CPU overhead for packet forwarding. We also implemented a QCN congestion control [7] using ServerSwitch. The experiments showed stable queue dynamics and that ServerSwitch can react to congestion in 23us.

ServerSwitch explores the design space of combining a high performance ASIC switching chip with limited programmability with a fully programmable multicore commodity server. Our key findings are as follows: 1) ServerSwitch shows that various packet forwarding schemes including source routing can be offloaded to the ASIC switching chip, hence resulting in small forwarding latency and zero CPU overhead. 2) With a low latency PCI-E interface, we can implement latency sensitive schemes such as QCN congestion control, using server CPU with a pure software approach. 3) The rich programmability and high performance provided by ServerSwitch can further enable new DCN services that need in-network data processing such as in-network caching [4].

The rest of the paper is organized as follows. We elaborate the design goals in § 2. We then present the architecture of ServerSwitch and our design choices in § 3. We illustrate the software, hardware, and API implementations in § 4. § 5 discusses how we use ServerSwitch to implement two real DCN designs, § 6 evaluates the platform with micro benchmarks and real DCN implementations. We discuss ServerSwitch limitations and 10G ServerSwitch in § 7. Finally, we present related work in § 8 and conclude in § 9.

2 Design Goals

As we have discussed in § 1, the goal of this paper is to design and implement a programmable and high performance DCN platform for existing and future DCN designs. Specifically, we have following design goals. First, on the data plane, the platform should provide a packet forwarding engine that is both programmable and achieves high-performance. Second, the platform needs to support new routing and signaling, flow/congestion control designs in the control plane. Third, the platform enables new DCN services (*e.g.*, in-network caching) by providing advanced in-network packet processing. To achieve these design goals, the platform needs to provide flexible programmability and high performance in both the data and control planes. It is highly desirable that the platform be easy to use and implemented in pure commodity and low cost silicon, which will ease the adoption of this platform in a real world product environment. We elaborate on these goals in detail in what follows.

Programmable packet forwarding engine. Packet forwarding is the basic service provided by a switch or router. Forwarding rate (packet per second, or PPS) is one of the most important metrics for network device evaluation. Current Ethernet switches and IP routers can offer line-rate forwarding for various packet sizes. However, recent DCN designs require a packet forwarding engine that goes beyond traditional destination MAC or IP address based forwarding. Many new DCN designs embed network topology information into server addresses and leverage this topology information for packet forwarding and routing. For example, PortLand [22] codes its fat-tree topology information into device MAC addresses and uses Longest Prefix Matching (LPM) over its PMAC (physical MAC) for packet forwarding. BCube uses source routing and introduces an NHI (Next Hop Index, §7.1 of [13]) to reduce routing path length by leveraging BCube structural information. We expect that more DCN architectures and topologies will appear in the near future. These new designs call for a programmable packet forwarding engine which can handle various packet forwarding schemes and packet formats.

New routing and signaling, flow/congestion control support. Besides the packet forwarding functions in the data plane, new DCN designs also introduce new control and signaling protocols in the control plane. For example, to support the new addressing scheme, switches in PortLand need to intercept the ARP packets, and redirect them to a Fabric Manager, which then replies with the PMAC of the destination server. BCube uses adaptive routing. When a source server needs to communicate with a destination server, the source server sends probing packets to probe the available bandwidth of multiple edge-disjoint paths. It then selects the path with the highest available bandwidth. The recent proposed QCN switches sample the incoming packets and send back queue and congestion information to the source servers. The source servers then react to the congestion information by increasing or decreasing the sending rate. All these functionalities require the switches to be able to filter and process these new control plane messages. Control plane signaling is time critical and sensitive to latency. Hence switches have to process these control plane messages in real time. Note that current switches/routers do offer the ability to process the control plane messages with their embedded CPUs. However, their CPUs mainly focus on management functions and are generally lack of the ability to process packets with high throughput and low latency.

New DCN service support by enabling in-network packet processing. Unlike the Internet which consists of many ISPs owned by different organizations, data centers are owned and administrated by a single operator. Hence we expect that technology innovations will be adopted faster in the data center environment. One such innovation is to introduce more intelligence into data center networks by enabling in-network traffic processing. For example, CamCube [4] proposed a cache service by introducing packet filtering, processing, and caching in the network. We can also introduce switch-assisted reliable multicast [18, 8] in DCN, as discussed in [26]. For an in-network packet processing based DCN service, we need the programmability such as arbitrary packet modification, processing and caching, which is much more than the programmability provided by the programmable packet forwarding engine in our first design goal. More importantly, we need low overhead, line-rate data processing, which may reach several to tens of Gb/s.

The above design goals call for a platform which is programmable for both data and control planes, and it needs to achieve high throughput and low processing latency. Besides the programmability and high performance design goals, we have two additional requirements (or constraints) from the real world. First, the programmability we provide should be easy to use. Second, it is highly desirable that the platform is built from (inexpensive) commodity components (*e.g.*, merchandise chips). We believe that a platform based on commodity components has a pricing advantage over non-commodity, expensive ones. The easy-to-program requirement ensures the platform is easy to use, and the commodity constraint ensures the platform is amenable to wide adoption.

Our study revealed that none of the existing platforms meet all our design goals and the easy-to-program and commodity constraints. The pure software based approaches, *e.g.*, Click, have full and easy-to-use programmability, but cannot provide low latency packet processing and high packet forwarding rate. FPGA-based systems, *e.g.*, NetFPGA, are not as easy to program as the commodity servers, and their prices are generally high. For example, the price of Virtex-II Pro 50 used in NetFPGA is 1,180$ per chip for 100+ chip quantum listed on the Xilinx website. Openflow switches provide certain programmability for both forwarding and control functions. But due to the separation of switches and the controller, it is unclear how Openflow can be extended to support congestion control and in-network data processing.

We design ServerSwitch to meet the three design goals and the two practical constraints. ServerSwitch has a hardware part and a software part. The hardware part is a merchandise switching chip based NIC plus a commodity server. The ServerSwitch software manages the hardware and provides APIs for developers to program and control ServerSwitch. In the next section, we will describe the architecture of ServerSwitch, and how ServerSwitch meets the design goals and constraints.

3 Design

3.1 ServerSwitch Architecture

Our ServerSwitch architecture is influenced by progress and trends in ASIC switching chip and server technologies. First, though commodity switches are black boxes to their users, the switching chips inside (*e.g.*, from Broadcom, Fulcrum, and Marvell) are becoming increasingly programmable. They generally provide exact matching (EM) based on MAC addresses or MPLS tags, provide longest prefix matching (LPM) based on IP addresses, and have a TCAM (ternary content-addressable memory) table. Using this TCAM table, they can provide arbitrary field matching. Of course, the width of the arbitrary field is limited by the hardware, but is generally large enough for our purpose. For example, Broadcom Enduro series chips have a maximum width of 32 bytes, and Fulcrum FM3000 can match up to 78 bytes in the packet header [3]. Based on the matching result, the matched packets can then be programmed to be forwarded, discarded, duplicated (*e.g.*, for multicast purpose), or mirrored. Though the programmability is limited, we will show later that it is already enough for

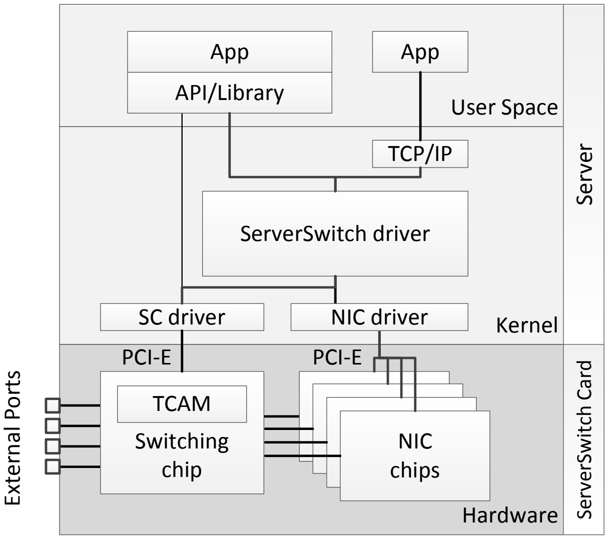


Figure 1: ServerSwitch architecture.

all packet forwarding functions in existing, and arguably, many future DCN designs.

Second, commodity CPU (*e.g.*, x86 and X64 CPUs) based servers now have a high-speed, low latency interface, *i.e.*, PCI-E, to connect to I/O subsystems such as a network interface card (NIC). Even PCI-E 1.0 X4 can provide 20Gbps bidirectional throughput and microsecond latency between the server CPU and NIC. Moreover, commodity servers are arguably the best programmable devices we currently have. It is very easy to write kernel drivers and user applications for packet processing with various development tools (*e.g.*, C/C++).

ServerSwitch then takes advantage of both commodity servers and merchandise switching chips to meet our design goals. Fig. 1 shows its architecture. The hardware part is an ASIC switching chip based NIC and a commodity server. The NIC and server are connected by PCI-E. From the figure, we can see there are two PCI-E channels. One is for the server to control and program the switching chip, the other is for data packet exchange between the server and switching chip.

The software part has a kernel and an application component, respectively. The kernel component has a switching chip (SC) driver to manage the commodity switching chip and an NIC driver for the NICs. The central part of the kernel component is a ServerSwitch driver, which sends and receives control messages and data packets through the SC and NIC drivers. The ServerSwitch driver is the place for various control messages, routing, congestion control, and various in-network packet processing. The application component is for developers. Developers use the provided APIs to interface with the ServerSwitch driver, and to program and control the switching chip.

Our ServerSwitch nicely fulfills all our design goals and meets the easy-to-program and commodity constraints. The switching chip provides a programmable packet forwarding engine which can perform packet matching based on flexible packet fields, and achieve full line rate forwarding even for small packet sizes. The ServerSwitch driver together with the PCI-E interface achieves low latency communication between the switching chip and server CPU. Hence various routing, signaling and flow/congestion controls can be well supported. Furthermore, the switch chip can be programmed to select specific packets into the server CPU for advanced processing (such as in-network caching) with high throughput. The commodity constraint is directly met since we use only commodity, inexpensive components in ServerSwitch. ServerSwitch is easy to use since all programming is performed using standard C/C++. When a developer introduces a new DCN design, he or she needs only to write an application to program the switching chip, and add any needed functions in the ServerSwitch driver.

The ability of our ServerSwitch is constrained by the abilities of the switching chip, the PCI-E interface, and the server system. For example, we may not be able to handle packet fields which are beyond the TCAM width, and we cannot further cut the latency between the switching chip and server CPU. In practice, however, we are still able to meet our design goals with these constraints. In the rest of this section, we will introduce the programmable packet forwarding engine, the software, and the APIs in detail.

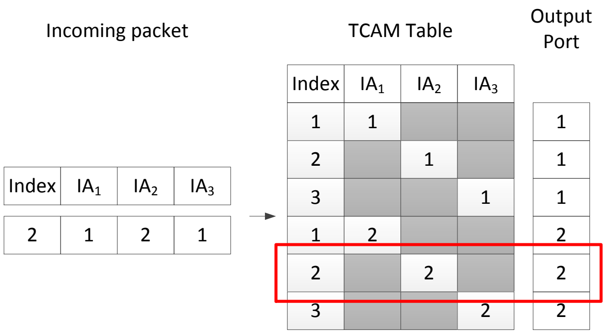
3.2 ASIC-based Programmable Packet Forwarding Engine

In this section, we discuss how existing Ethernet switching chips can be programmed to support various packet forwarding schemes.

There are three commonly used forwarding schemes in current DCN designs, *i.e.*, Destination Address (DA) based, tag-based, and Source Routing (SR) based forwarding. DA-based forwarding is widely adopted by Ethernet and IP networks. Tag-based forwarding decouples routing from forwarding which makes traffic engineering easier. SR-based forwarding gives the source server ultimate control of the forwarding path and simplifies the functions in forwarding devices. Table 1 summarizes the forwarding primitives and existing DCN designs for these three forwarding schemes. There are three basic primitives to forward a packet, *i.e.*, lookup key extraction, key matching, and header modification. Note that the matching criteria is independent of the forwarding schemes, *i.e.*, a forwarding scheme can use any matching criteria. In practice, two commonly used cri-

Scheme	Primitives			DCN Design
	Extract	Match	Modify	
DA-based	Direct	Any	No	Portland DCell
Tag-based	Direct	Any	SWAP/ POP/ PUSH	-
SR-based	Direct	Any	POP	VL2
	Indirect	Any	Change Index	BCube

Table 1: Forwarding schemes and primitives.



Index	IA_1	IA_2	IA_3
2	1	2	1

Index	IA_1	IA_2	IA_3	Output Port
1	1	▨	▨	1
2	▨	1	▨	1
3	▨	▨	1	1
1	2	▨	▨	2
2	▨	2	▨	2
3	▨	▨	2	2

Figure 2: Support indexed source routing using TCAM.

teria are EM and LPM. Next, we describe the three forwarding schemes in detail. We start from SR-based forwarding.

3.2.1 Source Routing based Forwarding using TCAM

For SR-based forwarding, there are two approaches depending on how the lookup key is extracted: indexed and non-indexed SR-based forwarding. In both approaches, the source fills a series of intermediate addresses (IA) in the packet header to define the packet forwarding path. For the non-Indexed Source Routing (nISR), the forwarding engine always uses the first IA for table lookup and pops it before sending the packet. For Indexed Source Routing (ISR), there is an index i to denote the current hop. The engine first reads the index, then extracts IA_i based on the index, and finally updates the index before sending the packet. We focus on ISR support in the rest of this subsection. We will discuss nISR support in the next subsection since it can be implemented as a form of tag-based forwarding.

ISR-based forwarding uses two steps for lookup key extraction. It first gets the index from a fixed location, and then extracts the key pointed by the index. However, commodity switching chips rarely have the logic to perform this two-step indirect lookup key extraction. In this paper, we design a novel solution by leveraging TCAM and turning this two-step key extraction into a single step key extraction. The TCAM table has many entries and each entry has a value and a mask. The mask is to set the masking bits ('care' and 'do-not-care' bits) for the value.

In our design, for each incoming packet, the forward engine compares its index field and all IA fields against the TCAM table. The TCAM table is set up as follows. For each TCAM entry, the index field (i) and the IA_i field pointed by the index are 'care' fields. All other IA fields are 'do-not-care' fields. Thus, a TCAM entry can simultaneously match both the index and the corresponding IA_i field. As both index and IA_i may vary, we enumerate all the possible combinations of index and IA values in the TCAM table. When a packet comes in, it will match one and only one TCAM entry. The action of that entry determines the operation on that matched packet.

Fig. 2 illustrates how the procedure works. The incoming packet has one index field and three IA fields. IA_2 is the lookup key for this hop. In the TCAM table, the white fields are the 'care' fields and the gray fields are the 'do-not-care' fields. Suppose there are two possible IA addresses and the maximum value of the index is three, there are 6 entries in the TCAM table. For this incoming packet, it matches the 5th entry where Index=2 and $IA_2 = 2$. The chip then directs the packet to output port 2. In § 5.1, we will describe the exact packet format based on our ServerSwitch.

This design makes a trade-off between the requirement of extra ASIC logic and the TCAM space. When there are n different IA values, the two-step indirect matching method uses n lookup entries, while this one-step method uses $n \times d$ entries where d is the maximum value of the index. d is always less than or equal to the network diameter. Modern switching chips have at least thousands of TCAM entries, so this one-step method works well in the DCN environment. For example, consider a medium sized DCN such as a three-level fat-tree in Portland. When using 48-port switches, there are 27,648 hosts. We can use 48 IA values to differentiate these 48 next hop ports. Since the diameter of the network is 6, the number of TCAM entries is $48 \times 6 = 288$, which is much smaller than the TCAM table size.

3.2.2 Destination and Tag-based Forwarding

As for the DA-based forwarding, the position of the lookup key is fixed in the packet header and the forwarding engine reads the key *directly* from the packet header. No lookup key modification is needed since the destination address is a globally unique id. However, the destination address can be placed anywhere in the packet header, so the engine must be able to perform matching on arbitrary fields. For example, Portland requires the switch to perform LPM on the destination MAC address, whereas DCell uses a self-defined header.

Tag-based routing also uses direct key extraction, but the tag needs to be modified on a per-hop basis since the tags have only local meaning. To support this routing scheme, the forwarding engine must support SWAP/POP/PUSH operations on tags.

Modern merchandise switching chips generally have a programmable parser, which can be used to extract arbitrary fields. The TCAM matching module is flexible enough to implement EM, LPM [25], and range matching. Hence, DA-based forwarding can be well supported.

For tag-based forwarding, many commodity switching chips for Metro Ethernet Network already support MPLS (multiple protocol label switching), which is the representative tag-based forwarding technology. Those chips support POP/PUSH/SWAP operations on the MPLS labels in the packet header. Hence we can support tag-based forwarding by selecting a switching chip with MPLS support. Further, by using tag stacking and POP operations, we can also support nISR-based forwarding. In such nISR design, the source fills a stack of tags to denote the routing path and the intermediate switches use the outermost tag for table lookup and then pops the tag before forwarding the packet.

3.3 Server Software

3.3.1 Kernel Components

The ServerSwitch driver is the central hub that receives all incoming traffic from the underlying ServerSwitch card. The driver can process them itself or it can deliver them to the user space for further processing. Processing them in the driver gives higher performance but requires more effort to program and debug. Meanwhile, processing these packets in user space is easy for development but scarifies performance. Instead of making a choice on behalf of users, ServerSwitch allows users to decide which one to use. For low rate control plane traffic where processing performance is not a major concern, *e.g.*, ARP packets, ServerSwitch can deliver them to user space for applications to process them. Since the applications need to send control plane traffic too, ServerSwitch provides APIs to receive packets from user-space applications to be sent down to the NIC chips. For those control plane packets with low latency requirement and high speed in-network processing traffic whose performance is a major concern, *e.g.*, QCN queue queries or data cache traffic, we can process them in the ServerSwitch driver.

The SC and NIC drivers both act as the data channels between the switching chip and the ServerSwitch driver. They receive packets from the device and deliver them to the ServerSwitch driver, and vice versa. The SC driver also provides an interface for the user library and the ServerSwitch to manipulate its registers directly, so both applications and the ServerSwitch driver can control the switching chip directly.

3.3.2 APIs

We design a set of APIs to control the switching chip and send/receive packets. The APIs include five categories as follows.

1. Set User Defined Lookup Key (UDLK): This API configures the programmable parser in the switching chip by setting the *i*-th UDLK. In this API, the UDLK can be fields from the packet header as well as meta-data, *e.g.*, the incoming port of a packet. We use the most generic form to define packet header fields, *i.e.*, the byte position of the desired fields. In the following example, we set the destination MAC address (6 bytes, B0-5) as the first UDLK. We can also combine meta-data (*e.g.*, incoming port) and non-consecutive byte range to define a UDLK, as shown in the second statement which is used for BCube (§ 5.1).

```
API:
  SetUDLK(int i, UDLK udlk)
Example:
  SetUDLK(1, (B0-5))
  SetUDLK(2, (INPORT, B30-33, B42-45))
```

2. Set Lookup Table: There are several lookup tables in the switching chip, a general purpose TCAM table, and protocol specific lookup tables for Ethernet, IP, and MPLS. This API configures different lookup tables denoted by *type*, and sets the *value*, *mask* and *action* for the *i*-th entry. The mask is NULL when the lookup table is an EM table. The *action* is a structure that defines the actions to be taken for the matched packets, *e.g.*, directing the packets to a specified output port, performing pre-defined header modifications, etc. For example, for MPLS the modification actions can be Swap/Pop/Push. The *iudlk* is the index of UDLK to be compared. *iudlk* is ignored for the tables that do not support UDLK.

In the following example, the statement sets the first TCAM entry and compares the destination MAC address (the first UDLK) with the value field (000001020001, *i.e.*, 00:00:01:02:00:01) using mask (FFFFFF000000). This statement is used to perform LPM on dest MAC for PortLand. Consequently, all matching packets are forwarded to the third virtual interface.

```
API:
  SetLookupTable(int type, int i,
    int iudlk, char *value, char* mask,
    ACTION *action)
Example:
  SetLookupTable(TCAM, 1,
    1, "000001020001", "FFFFFF000000",
    {act=REDIRECT_VIF, vif=3})
```

3. Set Virtual Interface Table: This API sets up the *i*-th virtual interface entry which contains destination and source MAC addresses as well as the output port. The MAC addresses are used to replace the original MACs in the packet when they are not NULL.

For example, the following command sets up the third virtual interface to deliver packets to output port 2. Meanwhile, the destination MAC is changed to the given value (001F29D417E8) accordingly. The edge switches in Portland need such functionality to change PMAC back to the original MAC (§3.2 in [22]).

```
API:
  SetVIfTable(int i, char *dmac,
        char *smac, int oport)
Example:
  SetVIfTable(3, "001F29D417E8", NULL, 2)
```

4. Read/Write Registers: There are many statistic registers in switching chip, *e.g.*, queue length and packet counters, and registers to configure the behaviors of the switching chip, *e.g.*, enable/disable L3 processing. This API is to read and write those registers (specified by *regname*). As an example, the following command returns the queue length (in bytes) of output port 0.

```
API:
  int ReadRegister (int regname)
  int WriteRegister(int regname, int value)
Example:
  ReadRegister(OUTPUT_QUEUE_BYTES_PORT0)
```

5. Send/Receive Packet: There are multiple NICs for sending and receiving packets. We can use the first API to send packet to a specific NIC port (*oport*). When we receive a packet, the second API also provides the input NIC port (*iport*) for the packet.

```
API:
  int SendPacket(char *pkt, int oport)
  int RecvPacket(char *pkt, int *iport)
```

4 Implementation

4.1 ServerSwitch Card

Fig. 3 shows the ServerSwitch card we designed. All chips used on the card are merchandise ASICs. The Broadcom switching chip BCM56338 has 8 Gigabit Ethernet (GE) ports and two 10GE ports [1]. Four of the GE ports connect externally and the other four GE ports connect to two dual GE port Intel 82576EB NIC chips. The two NIC chips are used to carry a maximum of 4Gb/s traffic between the switching chip and the server since the bandwidth of the PCI-E interface on 56338 is only 2Gb/s. The three chips connect to the server via a PCI-E switch PLX PEX8617. The effective bandwidth from the PEX8617 to BCM56338, the two NIC chips and the server are 2, 8, 8 and 8Gb/s (single direction). Since the maximum inbound or outbound traffic is 4Gb/s, PCI-E is not the bottleneck. The two 10GE XAUI ports are designed for interconnecting multiple ServerSwitch cards in one server chassis to create a larger non-blocking switching fabric with more ports. Each ServerSwitch card costs less than 400$ when manufactured in 100 pieces. We expect the price can be cut to 200$ for a quantity of 10K. The power consumption of ServerSwitch is 15.4W when all 8 GE ports are idle, and is 15.7W when all of them carry full speed traffic.

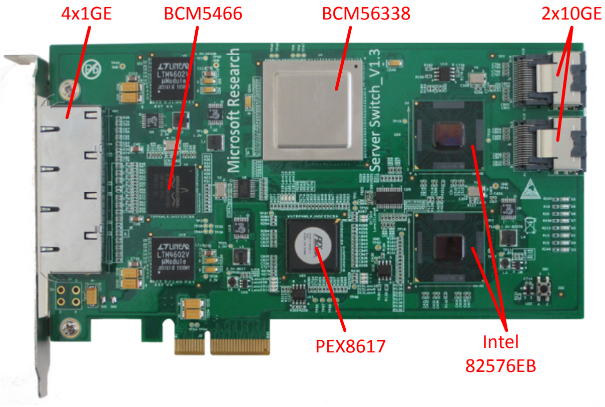


Figure 3: ServerSwitch card.

Fig. 4 shows the packet processing pipeline of the switching chip, which has three stages. First, when the packets go into the switching chip, they are directed to a programmable parser and a classifier. The classifier then directs the packets to one of the protocol specific header parsers. The Ethernet parser extracts the destination MAC address (DMAC), the IP parser extracts the destination IP address (DIP), the MPLS parser extracts the MPLS label and the Prog parser can generate two different UDLKs. Each UDLK can contain any aligned four 4-byte blocks from the first 128 bytes of the packet, and some meta-data of the packet.

Next, the DMAC is sent to the EM(MAC) matching module, the DIP to both the LPM and EM(IP) matching modules, the MPLS label to the EM(MPLS) module, and the UDLK to the TCAM. Each TCAM entry can select one of the two UDLKs to match. The matchings are performed in parallel. The three matching modules (EM, LPM, TCAM) result in an index into the interface table, which contains the output port, destination and source MAC. When multiple lookup modules match, the priority of their results follows TCAM > EM > LPM.

Finally, the packet header is modified by the L3 and L2 modifiers accordingly. The L3 modifier changes the L3 header, *e.g.*, IP TTL, IP checksum and MPLS label. The

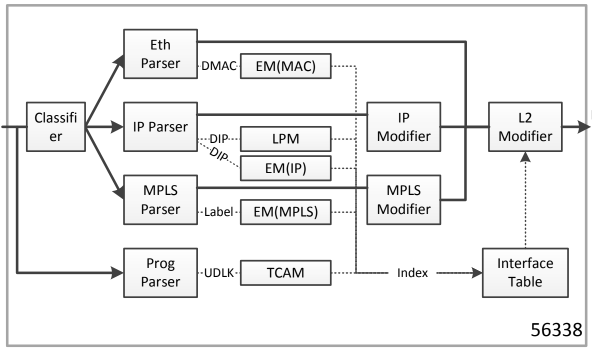


Figure 4: Packet processing pipeline in Broadcom 56338.

L2 modifier can use the MAC addresses in the interface table to replace the original MAC addresses.

The size of EM tables for MAC, IPv4 and MPLS are 32K, 8K and 4K entries, respectively. The LPM for IPv4 and the TCAM table have 6144 and 2K entries, respectively. The interface table has 4K entries. All these tables, the Prog Parser and the behaviors of the modifiers are programmable.

4.2 Kernel Drivers

We have developed ServerSwitch kernel drivers for Windows Server 2008 R2. As shown in Fig. 1, it has components as follows.

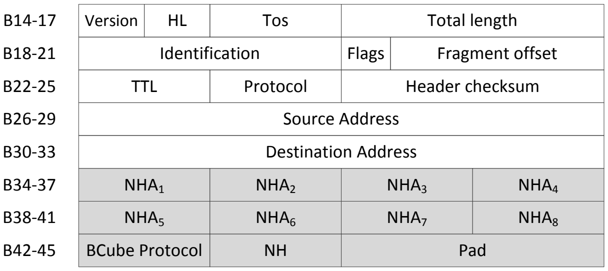
Switching Chip Driver. We implemented a PCI-E driver based on Broadcom's Dev Kits. The driver has 2670 lines of C code. It allocates a DMA region and maps the chip's registers into memory address using memory-mapped I/O (MMIO). The driver can deliver received packets to the ServerSwitch driver, and send packets to hardware. The ServerSwitch driver and user library can access the registers and thus control the switching chip via this SC driver.

NIC Driver. We directly use the most recent Intel NIC driver binaries.

ServerSwitch Driver. We implemented the ServerSwitch driver as a Windows NDIS MUX driver. It has 20719 lines of C code. The driver exports itself as a virtual NIC. It binds the TCP/IP stack on its top and the Intel NIC driver and the SC driver at its bottom. The driver uses `IRP` to send and receive packets from the user library. It can also deliver the packets to the TCP/IP stack. The ServerSwitch driver provides a kernel framework for developing various DCN designs.

4.3 User Library

The library is based on the Broadcom SDK. The SDK has 3000K+ lines of C code and runs only on Linux and VxWorks. We ported this SDK to Subsystem for UNIX-based Applications (SUA) on Windows Server 2008 [2]. At the bottom of the SDK, we added a library to interact with our kernel driver. We then developed ServerSwitch APIs over the SDK.



B14-17	Version	HL	Tos	Total length	
B18-21	Identification			Flags	Fragment offset
B22-25	TTL		Protocol	Header checksum	
B26-29	Source Address				
B30-33	Destination Address				
B34-37	NHA$_1$		NHA$_2$	NHA$_3$	NHA$_4$
B38-41	NHA$_5$		NHA$_6$	NHA$_7$	NHA$_8$
B42-45	BCube Protocol		NH	Pad	

Figure 5: BCube header on the ServerSwitch platform.

5 Building with the ServerSwitch Platform

In this section, we use ServerSwitch to implement several representative DCN designs. First, we implement BCube to illustrate how indexed source routing is supported in the switching chip of ServerSwitch. In our BCube implementation, BCube packet forwarding is purely carried out in hardware. Second, we show our implementation of QCN congestion control. Our QCN implementation demonstrates that our ServerSwitch can generate low latency control messages using the server CPU. We discuss how ServerSwitch can support other DCN designs in the Appendix.

5.1 BCube

BCube is a server centric DCN architecture [13]. BCube uses adaptive source routing. Source servers probe multiple paths and select the one with the highest available bandwidth. BCube defines two types of control messages, for neighbor discovery (ND) and available bandwidth query (ABQ) respectively. The first one is for servers to maintain the forwarding table. The second one is used to probe the available bandwidth of the multiple parallel paths between the source and destination.

Our ServerSwitch is an ideal platform for implementing BCube. For an intermediate server in BCube, our ServerSwitch card can offload packet forwarding from the server CPU. For source and destination servers, our ServerSwitch card can achieve k:1 speedup using k NICs connected by BCube topology. This is because in our design the internal bandwidth between the server and the NICs is equal to the external bandwidth provided by the multiple NICs, as we show in Figure 1.

Fig. 5 shows the BCube header we use. It consists of an IP header and a private header (gray fields). We use this private header to implement the BCube header. We use an officially unassigned IP protocol number to differentiate the packet from normal TCP/UDP packets. In the private header, the BCube protocol is used to identify control plane messages. NH is the number of valid NHA fields. It is used by a receiver to construct a reverse path to the sender. There are 8 1-byte Next Hop Address (NHA) fields, defined in BCube for indexed source routing. Different from NHA in the original BCube header design, NHAs are filled in reverse order in our private header. NHA$_1$ is now the lookup key for the last hop. This implementation adaption is to obtain an automatic index counter by the hardware. We observe that for a normal IP packet, its TTL is automatically decreased after one hop. Therefore, we overload the TTL field in the IP header as the index field for NHAs. This is the reason why we store NHAs in reverse order.

We implemented a BCube kernel module in the ServerSwitch driver and a BCube agent at the user-level. The kernel module implements data plane functionalities. On the receiving direction, it delivers all received control messages to the user-level agent for processing. For any received data packets, it removes their BCube headers and delivers them to the TCP/IP stack. On the sending direction, it adds the BCube header for the packets from the TCP/IP stack and sends them to the NICs.

The BCube agent implements all control plane functionalities. It first sets up the ISR-based forwarding rules and the packet filter rules in the switching chip. Then, it processes the control messages. When it receives an ND message, it updates the interface table using `SetVIfTable`. It periodically uses `ReadRegister` to obtain traffic volume from the switching chip and calculates the available bandwidth for each port. When it receives an ABQ message, it encodes the available bandwidth in the ABQ message, and sends it to the next hop.

Fig. 6 shows the procedure to initialize the switching chip for BCube, using the ServerSwitch API. Line 1 sets a 12-byte UDLK$_1$ for source routing, including TTL (`B22`) and the NHA fields (`B34-41`). Line 2 sets another 9-byte UDLK$_2$ for packet filtering, including incoming port number (`INPORT`), IP destination address (`B30-33`) and BCube protocol (`B42`). The `INPORT` occupies 1-byte field. Lines 5-18 set the ISR-based TCAM table. Since every NHA denotes a neighbor node with a destination MAC and corresponding output port, line 8 sets up one interface entry for one NHA value. Lines 13-16 sets up a TCAM entry to match the TTL and its corresponding NHA in UDLK$_1$. Since the switch discards the IP packets whose TTL ≤ 1, we use TTL = 2 to denote NHA$_1$. Lines 21-38 filter packets to the server. Since the switching chip has four external (0-3) and four internal ports (4-7), we filter the traffic of an external port to a corresponding internal port, *i.e.*, port 0→4, 1→5, 2→6 and 3→7. Line 23 sets `action` to direct the packets to port $4 \sim 7$ respectively. Lines 27-29 match those packets whose destination BCube address equals the local BCube address in UDLK$_2$. Lines 32-37 match BCube control plane messages, *i.e.*, ND and ABQ, in UDLK$_2$. In our switching chip, when a packet matches multiple TCAM entries, the entry with the highest index will win. Therefore, in our BCube implementation, entries for control plane messages have higher priority than the other ones.

```
1:  SetUDLF(1, (B22-25, B34-41));
2:  SetUDLF(2, (INPORT, B30-33, B42-45));
3:
4:  // setup ISR-based forwarding table
5:  j = 0;
6:  foreach nha in (all possible NHA values)
7:  {
8:    SetVifTable(nha, dstmac, srcmac, oport);
9:    for (index = 0; index < 8; index++)
10:   {
11:     // val[0] matches B22 (TTL) in UDLF_1
12:     // val[4:11] matches B34-41 (NHAs) in UDLF_1
13:     val[0]   = index+2; mask[0] = 0xff;
14:     val[4+index] = nha; mask[4+index] = 0xff;
15:     action.act = REDIRECT_VIF;  action.vif = nha;
16:     SetLookupTable(TCAM,j++,1,val,mask,&action);
17:   }
18: }
19:
20: // setup filter to server
21: for (i = 0; i < 4; i++)
22: {
23:   action.act = REDIRECT_PORT; action.port = 4 + i
24:   // filter packets that are sent to localhost
25:   // val[0] matches INPORT in UDLF_2
26:   // val[1:4] match B30-33 (IP dst addr) in UDLF_2
27:   val[0]   = i; mask[0] = 0xff
28:   val[1:4] = my_bcube_id; mask[1:4] = 0xffffffff;
29:   SetLookupTable(TCAM,j++,2,val,mask,&action);
30:   // filter control plane packets
31:   // val[5] matches B42 (BCube prot) in UDLF_2
32:   val[0] = i;   mask[0] = 0xff;
33:   val[5] = ND;  mask[5] = 0xff;
34:   SetLookupTable(TCAM,j++,2,val,mask,&action);
35:   val[0] = i;   mask[0] = 0xff;
36:   val[5] = ABQ; mask[5] = 0xff;
37:   SetLookupTable(TCAM,j++,2,val,mask,&action);
38: }
```

Figure 6: Pseudo TCAM setup code for BCube.

5.2 Quantized Congestion Control (QCN)

QCN is a rate-based congestion control algorithm for the Ethernet environment [7]. The algorithm has two parts. The Switch or Congestion Point (CP) adaptively samples incoming packets and generates feedback messages addressed to the source of the sampled packets. The feedback message contains congestion information at the CP. The Source or Reaction Point (RP) then reacts based on the feedback from the CP. See [7] for QCN details. The previous studies of QCN are based on simulation or hardware implementation.

We implemented QCN on the ServerSwitch platform as shown in Fig. 7. The switching chip we use cannot

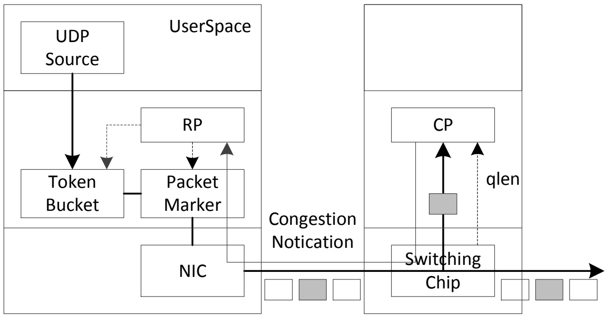


Figure 7: QCN on the ServerSwitch platform.

adaptively sample packets based on the queue length, so we let the source mark packets adaptively and let the ServerSwitch switching chip mirror the marked packets to the ServerSwitch CPU. When the ServerSwitch CPU receives the marked packets, it immediately reads the queue length from the switching chip and sends the Congestion Notification (CN) back to the source. When the source receives the CN message, it adjusts its sending rate and marking probability.

We implemented the CP and RP algorithms in ServerSwitch and end-host respectively based on the most recent QCN Pseudo code V2.3 [24]. In order to minimize the response delay, the CP module is implemented in the ServerSwitch driver. The CP module sets up a TCAM entry to filter marked packets to the CPU. On the end-host, we implemented a token bucket rate limiter in the kernel to control the traffic sending rate at the source.

6 Evaluation

Our evaluation has two parts. In the first part, we show micro benchmarks for our ServerSwitch. We evaluate its performance on packet forwarding, register read/write, and in-network caching. For micro benchmark evaluation, we connect our ServerSwitch to a NetFPGA card and use NetFPGA to generate traffic. In the second part, we implement two DCN designs, namely BCube and QCN, using ServerSwitch. We build a 16-server BCube$_1$ network to run BCube and QCN experiments. We currently build only two ServerSwitch cards. As shown in Fig. 8, the two gray nodes are equipped with ServerSwitch cards, they use an ASUS motherboard with Intel Quad Core i7 2.8GHz CPU. The other 14 servers are Dell Optiplex 755 with 2.4Ghz dual core CPU. The switches are 8-port DLink DGS-1008D GE switches.

6.1 Micro Benchmarks

We directly connect the four GE ports of one ServerSwitch to the four GE ports of one NetFPGA, and use

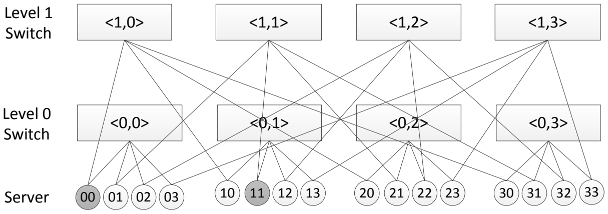


Figure 8: BCube$_1$ testbed.

the NetFPGA-based packet generator to generate line-rate traffic to evaluate the packet forwarding performance of ServerSwitch. We record the packet send and receive time using NetFPGA to measure the forwarding latency of ServerSwitch. The precision of the timestamp recorded by NetFPGA is $8ns$.

Forwarding Performance. Fig. 9 compares the forwarding performance of our ServerSwitch card and a software-based BCube implementation using an ASUS quad core server. In the evaluation, we use NetFPGA to generate 4GE traffic. The software implementation of the BCube packet forwarding is very simple. It uses NHA as an index to get the output port. (See §7.2 in [13] for more details) As we can see, there is a very huge performance gap between these two approaches. For ServerSwitch, there is no packet drop for any packet sizes, and the forwarding delay is small. The delays for 64 bytes and 1514 bytes are 4.3us and 15.6us respectively, and it grows linearly with the packet size. The slope is 7.7ns per byte, which is very close to the transmission delay of one byte over a GE link. The curve suggests the forwarding delay is a 4.2us fixed processing delay plus the transmission delay. For software forwarding, the maximum PPS achieved is 1.73Mpps and packets get dropped when the packet size is less than or equal to 512 bytes. The CPU utilization for 1514 byte is already 65.6%. Moreover, the forwarding delay is also much larger than that of ServerSwitch. This experiment suggests that a switching chip does a much better job for packet forwarding, and that using software for 'simple' packet forwarding is not efficient.

Register Read/Write Performance. Certain applications need to read and write registers of the switching chip frequently. For example, our software-based QCN needs to frequently read queue length from the switching chip. In this test, we continuously read/write a 32-bit register 1,000,000 times, and the average R/W latency of one R/W operation is 6.94/4.61us. We note that the latency is larger than what has been reported before (around $1us$) [20]. This is because [20] measured the latency of a single MMIO R/W operation, whereas our registers are not mapped but are accessed indirectly via several mapped registers. In our case, a read operation consists of four MMIO write and three MMIO read

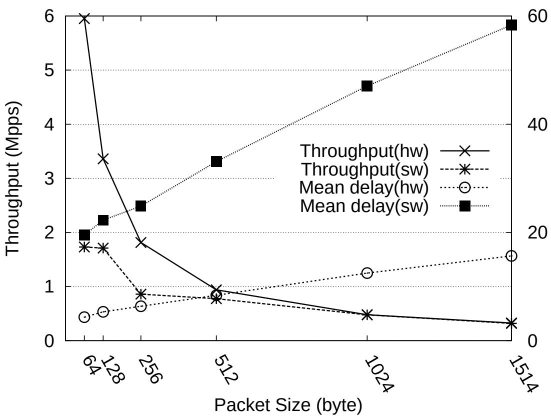


Figure 9: Packet forwarding performance.

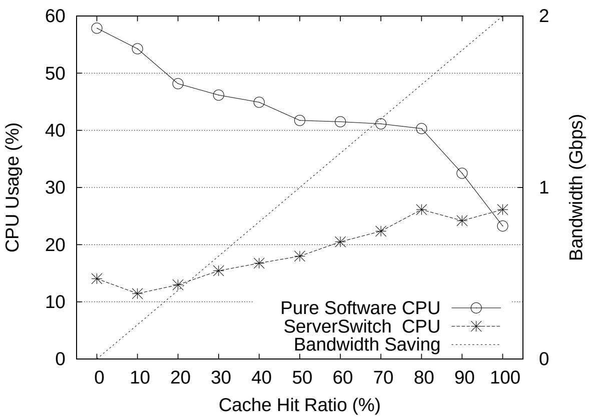


Figure 10: CPU utilization for in-network caching.

operations. We note that the transmission delay of one 1514-bytes packet over 1GE link is $12us$, so the read operation of our ServerSwitch can be finished within the transmission time of one packet.

In-network Caching Performance. We show that ServerSwitch can be used to support in-network caching. In this experiment, ServerSwitch uses two GbE ports to connect to NetFPGA A and the other two ports to NetFPGA B. NetFPGA A sends request packets to B via ServerSwitch. When B receives one request, it replies with one data packet. The sizes of request and reply are 128 and 1514 bytes, respectively. Every request or reply packet carries a unique ID in its packet header. When ServerSwitch receives a request from A, the switching chip performs an exact matching on the ID of the request. A match indicates that the ServerSwitch has already cached the response packet. The request is then forwarded to the server CPU which sends back the cached copy to A. When there is no match, the request is forwarded to B, and B sends back the response data. ServerSwitch also oversees the response data and tries to cache a local copy. The request rate per link is 85.8Mb/s, so the response rate per link between ServerSwitch and A is 966Mb/s. Since one NetFPGA has 4 ports, we use one NetFPGA to act as both A and B in the experiment.

We vary the cache hit ratio at ServerSwitch and measure the CPU overhead of the ServerSwitch. In-network caching increases CPU usage at ServerSwitch, but saves bandwidth between B and ServerSwitch. In our toy network setup, a $x\%$ cache hit rate directly results in $x\%$ bandwidth saving between B and ServerSwitch (as shown in Fig. 10). In a real network environment, we expect the savings will be more significant since we can save more bandwidth for multi-hop cases.

Fig. 10 also shows the CPU overhead of the ServerSwitch for different cache hit ratios. Of course, the higher the cache hit ratio, the more bandwidth we can save and the more CPU usage we need to pay. Note that in Fig. 10, even when the cache hit ratio is 0, we still have a cost of 14% CPU usage. This is because ServerSwitch needs to do caching for the 1.9Gbps response traffic from B to ServerSwitch. Fig. 10 also includes the CPU overhead of a pure software-based caching implementation. Our result clearly shows that our ServerSwitch significantly outperforms pure software-based caching.

6.2 ServerSwitch based BCube

In this experiment, we set up two TCP connections C1 and C2 between servers 01 and 10. The two connections use two parallel paths, P1 $\{01, 00, 10\}$ for C1 and P2 $\{01, 11, 10\}$ for C2, respectively. We run this experiment twice. First, we configure 00 and 11 to use the ServerSwitch cards for packet forwarding. Next, we configure them to use software forwarding. In both cases, the total throughput is 1.7Gbps and is split equally into the two parallel paths. When using ServerSwitch for forwarding, both 00 and 11 use zero CPU cycles. When using software forwarding, both servers use 15% CPU cycles. Since both servers have a quad core CPU, 15% CPU usage equals 60% for one core.

6.3 ServerSwitch based QCN

In this experiment, we configure server 00 to act as a QCN-enabled node. We use `iperf` to send UDP traffic from server 01 to 10 via 00. The sending rate of `iperf` is limited by the traffic shaper at 01. When there is congestion on level-1 port of 00, 00 sends CN to 01. We use the QCN baseline parameters [7] in this experiment.

Fig. 11 shows the throughput of the UDP traffic and the output queue length at server 00. When we start the UDP traffic, level 1 port is 1Gb/s. There is no congestion and the output queue length is zero. At time 20s, we limit level 1 port at 00 to 200Mb/s, the queue immediately builds up and causes 00 to send CN to the source. The source starts to use the QCN algorithm to adjust its traffic rate in order to maintain the queue length around Q_EQ which is 50KB in this experiment. We can see that the sending rate decreases to 200Mb/s very fast. And then we increase the bandwidth by 200Mb/s every 20 seconds. Similarly, the source adapts quickly to the new bandwidth. As shown in the figure, the queue length fluctuates around Q_EQ. This shows that this software-based implementation performs good congestion control. The rate of queue query packets processed by node 00 is very low during the experiment, with maximum and mean values of 801 and 173 pps. Hence QCN message processing introduces very little additional CPU overhead. The total CPU utilization is smaller than 5%. Besides, there is no packet drop in the experiment, even at the point when we decrease the bandwidth to 200Mb/s. QCN therefore achieves lossless packet delivery. We have varied the Q_EQ from 25KB to 200KB and the results are similar.

The extra delay introduced by our software approach to generate a QCN queue reply message consists of three parts: directing the QCN queue query to the CPU, reading the queue register, and sending back the QCN queue reply. To measure this delay, we first measure time RTT_1 between the QCN query and reply at 01. Then we configure the switching chip to simply bounce the QCN query back to the source assuming zero delay response for hardware implementation. We measure the time RTT_2 between sending and receiving a QCN query at 01. RTT_1 - RTT_2 reflects the extra delay introduced by software. The packet sizes of the queue query and reply are both 64 bytes in this measurement. The average values of RTT_1 and RTT_2 are 41us and 18us based on 10,000 measurements. Our software introduces only 23us delay. This extra delay is tolerable since it is comparable to or smaller than the packet transmission delay for one single 1500-bytes in a multi-hop environment.

7 Discussion

Limitations of ServerSwitch. The current version of ServerSwitch has the following limitations: 1) Limited hardware forwarding programmability. The switching chip we use has limited programmability on header field modification. It supports only standard header modifications of supported protocols (*e.g.*, changing Ethernet MAC addresses, decreasing IP TTL, changing IP DSCP, adding/removing IP tunnel header, modifying MPLS header). Due to the hardware limitation, our implementation of index-based source routing has to re-interpret the IP TTL field. 2) Relatively high packet processing latency due to switching chip to CPU communication. For the packets that require 'real' per-packet processing such as congestion information calculation in XCP protocol, the switching chip must deliver them to the CPU for processing, which leads to higher latency. Hence ServerSwitch is not suitable for protocols that need real time per-packet processing such as XCP. 3) Restricted form factor and relatively low speed. At present, a ServerSwitch card provides only 4 GbE ports. Though it can be directly used for server-centric or hybrid designs, *e.g.*, BCube, DCell, and CamCube, we do not expect that the current ServerSwitch can be directly used for architectures that need a large number of switch ports (48-ports or more), *e.g.*, fat-tree and VL2. However, since 4 ServerSwitch cards can be connected together to provide 16 ports, we believe ServerSwitch is still a viable platform for system prototyping for such architectures.

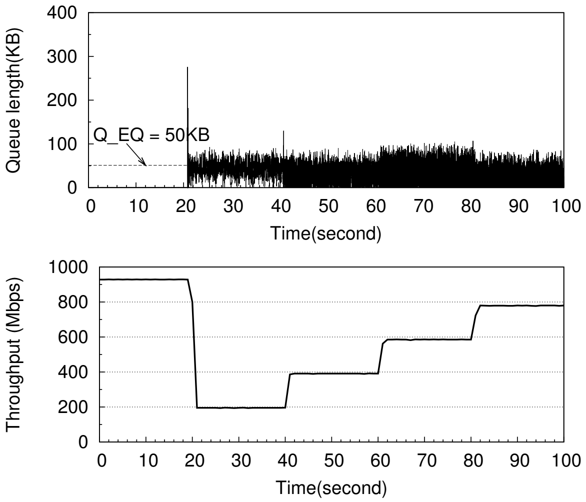


Figure 11: Throughput and queue dynamics during bandwidth change.

10GE ServerSwitch. Using the same hardware architecture, we can build a 10GE ServerSwitch. We need to upgrade the Ethernet switching chip, the PCI-E switching chip and the NIC chips. As for the Ethernet switching chip, 10GbE switching chips with 24x10GbE ports or more are already available from Broadcom, Fulcrum or Marvell. We can use two dual 10GbE Ethernet controller chips to provide a 40Gb/s data channel between the card and server CPU. Since we do not expect all traffic to be delivered to the CPU for processing, the internal bandwidth between the card and the server does not need to match the total external bandwidth. In this case, the number of external 10GE ports can be larger than four. We also need to upgrade the PCI-E switching chip to provide an upstream link with 40Gb/s bandwidth, which requires PCI-E Gen2 x8. Since the signal rate on the board is 10x faster than that in the current ServerSwitch, more hardware engineering effort will be needed to guarantee the

Signal Integrity (SI).

All the chips discussed above are readily available in the market. The major cost of such a 10GbE card comes from the 10GbE Ethernet switching chip, which has a much higher price than the 8xGbE switching chip. For example, a chip with 24 10GbE ports may cost about 10x that of the current one. The NIC chip and PCI-E switching chip cost about 2x∼3x than current ones. Overall, we expect the 10GE version card to be about 5x more expensive than the current 1GE version.

8 Related Work

OpenFlow defines an architecture for a central controller to manage OpenFlow switches over a secure channel, usually via TCP/IP. It defines a specification to manage the flow table inside the switches. Both OpenFlow and ServerSwitch aim towards a more programmable networking platform. Aiming to provide both programmability and high performance, ServerSwitch uses multiple PCI-E lanes to interconnect the switching chip and the server. The low latency and high speed of the channel enables us to harness the resources in a commodity server to provide both programmable control and data planes. With Openflow, however, it is hard to achieve similar functionalities due to the higher latency and lower bandwidth between switches and the controller.

Orphal provides a common API for proprietary switching hardware [21], which is similar to our APIs. Specifically, they also designed a set of APIs to manage the TCAM table. Our work is more than API design. We introduce a novel TCAM table based method for index-based source routing. We also leverage the resources of a commodity server to provide extra programmability.

Flowstream uses commodity switches to direct traffic to commodity servers for in-network processing [12]. The switch and the server are loosely coupled, *i.e.*, the server cannot directly control the switching chip. In ServerSwitch, the server and the switching chip are tightly coupled, which enables ServerSwitch to provide new functions such as software-defined congestion control which requires low-latency communication between the server and the switching chip.

Recently, high performance software routers, *e.g.*, RouteBricks [10] and PacketShader [16] have been designed and implemented. By leveraging multi-cores, they can achieve tens of Gb/s throughput. ServerSwitch is complementary to these efforts in that ServerSwitch tries to offload certain packet forwarding tasks from the CPU to a modern switching chip. ServerSwitch also tries to optimize its software to process low latency packets such as congestion control messages. At present, due to hardware limitations, ServerSwitch only provides 4x1GE ports. RouteBricks or PacketShader can certainly leverage a future 10GE ServerSwitch card to provide a higher throughput system, with a portion of traffic forwarded by the switching chip.

Commercial switches generally have an embedded CPU for switch management. More recently, Arista's 7100 series introduces the use of dual-core x86 CPU and provides APIs for programmable management plane processing. ServerSwitch differs from existing commodity switches in two ways: (1) The CPUs in commodity switches mainly focus on management functions, whereas ServerSwitch explores a way to combine the switching chip with the most advanced CPUs and server architecture. On this platform, the CPUs can process forwarding/control/management plane packets with high throughput and low latency. The host interface on the switching chip usually has limited bandwidth since the interface is designed for carrying control/management messages. ServerSwitch overcomes this limitation by introducing additional NIC chips for a high bandwidth, yet low latency channel between the switching chip and the server; (2) ServerSwitch tries to provide a common set of APIs to program the switch chip. The APIs are designed to be as universal as possible. Ideally, the API is the same no matter what kind of switching chip is used.

Ripcord [9] mainly focuses on the DCN control plane. It currently uses OpenFlow switches as its data plane. Our work is orthogonal to their work. We envision that they can also use ServerSwitch to support new DCN such as BCube, and to support more routing schemes such as source routing and tag-based routing.

9 Conclusion

We have presented the design and implementation of ServerSwitch, a programmable and high performance platform for data center networks. ServerSwitch explores the design space of integrating a high performance, limited programmable ASIC switching chip with a powerful, fully programmable multicore commodity server.

ServerSwitch achieves easy-to-use programmability by using the server system to program and control the switching chip. The switching chip can be programmed to support a flexible packet header format and various user defined packet forwarding designs with line-rate without the server CPU intervening. By leveraging the low latency PCI-E interface and efficient server software design, we can implement software defined signaling and congestion control in the server CPU with low CPU overhead. The rich programmability provided by ServerSwitch can further enable new DCN services that need in-network data processing such as in-network caching.

We have built a ServerSwitch card and a whole ServerSwitch software stack. Our implementation experiences demonstrate that ServerSwitch can be fully constructed from commodity, inexpensive components. Our development experiences further show that ServerSwitch is easy to program, using the standard C/C++ language and development tool chains. We have used our ServerSwitch platform to construct several recently proposed DCN designs, including new DCN architectures BCube and PortLand, congestion control algorithm QCN, and DCN in-network caching service.

Our software API currently focuses on lookup table programmability and queue information query. Current switching chips also provide advanced features such as queue and buffer management, access control, and priority and fair queueing scheduling. We plan to extend our API to cover these features in our future work. We also plan to upgrade the current 1GE hardware to 10G in the next version. We expect that ServerSwitch may be used for networking research beyond DCN (*e.g.*, enterprise networking). We plan to release both the ServerSwitch card and the software package to the networking research community in the future.

Acknowledgements

We thank Xiongfei Cai and Hao Zhang for their work on the initial ServerSwitch hardware design, the members of the Wireless and Networking Group and Zheng Zhang at Microsoft Research Asia for their support and feedback.

References

[1] `http://www.broadcom.com/collateral/pb/56330-PB01-R.pdf`.

[2] `http://www.suacommunity.com/SUA.aspx`.

[3] FM3000 Policy Engine, 2008. `http://www.fulcrummicro.com/documents/applications/FM3000_Policy_Engine.pdf`.

[4] Abu-Libdeh, H., Costa, P., Rowstron, A., O'Shea, G., and Donnelly, A. Symbiotic Routing in Future Data Centers. In *ACM SIGCOMM* (2010).

[5] Al-Fares, M., Loukissas, A., and Vahdat, A. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM* (2008).

[6] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI* (2010).

[7] Alizadeh, M., Atikoglu, B., Kabbani, A., Lakshmikantha, A., Pan, R., Prabhakar, B., and Seaman, M. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *46th Annual Allerton Conference on Communication, Control, and Computing,* (2008).

[8] Calderon, M., Sedano, M., Azcorra, A., and Alonsa, C. Active Network Support for Multicast Applications. *Network, IEEE 12*, 3 (may. 1998), 46 –52.

[9] Casado, M., et al. Ripcord: a Module Platform for Data Center Networking. Tech. Rep. UCB/EECS-2010-93, Univeristy of California at Berkeley, 2010.

[10] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SOSP* (2009).

[11] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D., Patel, P., and Sengupta, S. VL2: a Scalable and Flexible Data Center Network. In *ACM SIGCOMM* (2009).

[12] Greenhalgh, A., Huici, F., Hoerdt, M., Papadimitriou, P., Handley, M., and Mathy, L. Flow Processing and the Rise of Commodity Network Hardware. *SIGCOMM Comput. Commun. Rev. 39*, 2 (2009), 20–26.

[13] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *ACM SIGCOMM* (2009).

[14] Guo, C., Lu, G., Wang, H. J., Yang, S., Kong, C., Sun, P., Wu, W., and Zhang, Y. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *ACM CoNext* (2010).

[15] Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., and Lu, S. DCell: A Scalable and Fault Tolerant Network Structure for Data Centers. In *ACM SIGCOMM* (2008).

[16] Han, S., Jang, K., Park, K., and Moon, S. PacketShader: a GPU-Accelerated Software Router. In *ACM SIGCOMM* (2010).

[17] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. *ACM Transactions on Computer Systems* (August 2000), 263–297.

[18] Lehman, L., Garland, S., and Tennenhouse, D. Active Reliable Multicast. In *IEEE INFOCOM* (1998).

[19] Lu, G., Guo, C., Li, Y., Zhou, Z., Yuan, T., Wu, H., Xiong, Y., Gao, R., and Zhang, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. Tech. Rep. MSR-TR-2011-24, Microsoft Research, 2011.

[20] Miller, D. J., Watts, P. M., and Moore, A. W. Motivating Future Interconnects: A Differential Measurement Analysis of PCI Latency. In *ACM/IEEE ANCS* (2009).

[21] Mogul, J. C., Yalagandula, P., Tourrilhes, J., McGeer, R., Banerjee, S., Connors, T., and Sharma, P. API Design Challenges for Open Router Platforms on Proprietary Hardware. In *ACM HotNets-VII* (2008).

[22] Mysore, R. N., et al. PortLand: a Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *ACM SIGCOMM* (2009).

[23] Naous, J., Gibb, G., Bolouki, S., and McKeown, N. NetFPGA: Reusable Router Architecture for Experimental Research. In *PRESTO* (2008).

[24] Pan, R. QCN Pseudo Code. `http://www.ieee802.org/1/files/public/docs2009/au-rong-qcn-serial-hai-v23.pdf`.

[25] Shah, D., and Gupta, P. Fast Updating Algorithms for TCAMs. *IEEE Micro 21*, 1 (2001), 36–47.

[26] Shieh, A., Kandula, S., and Sirer, E. G. Sidecar: Building Programmable Datacenter Networks without Programmable Switches. In *ACM HotNets* (2010).

A More DCN Designs

In Portland [22], the edge switches intercept the ARP request and respond with PMAC. The switches use LPM on PMAC to forward the packet. The last hop switches convert the PMAC to AMAC. ServerSwitch can intercept ARP by matching its Ethernet type field and deliver it to the CPU for processing. It can also use TCAM to perform LPM on the destination MAC address.

In fat-tree [5], the switches perform two-level lookup on IP destination address. The paper describes an implementation of this lookup algorithm using TCAM. Thus, using ServerSwitch to implement fat-tree is straightforward.

DCell [15] uses a self-defined header between the Ethernet and IP header. Servers use a 32-bit destination DCell address to forward the packet. Similar to BCube, we can use the IP header on our platform and put the DCell address into the destination IP address. The DCell address is hierarchical but its subnet size is not exactly 2^k, so we can join multiple smaller LPM entries to match a single DCell subnet. The DCell header has a Proxy Flag bit. When it is set, the server should use the Proxy DCell address to forward the packet. We can create a TCAM entry to match both the Proxy Flag and the Proxy DCell address. When both of them match, the chip overrides the decision based on the destination address and forwards the packet based on this TCAM entry.

These three extra examples show the generality and flexibility of the ServerSwitch packet forwarding engine.