CAFE: A Configurable pAcket Forwarding Engine for Data Center Networks

Guohan Lu
Microsoft Research Asia
Beijing, China
lguohan@microsoft.com

Yunfeng Shi*
Peking University
Beijing, China
shiyunfeng@gmail.com

Chuanxiong Guo
Microsoft Research Asia
Beijing, China
chguo@microsoft.com

YongguangZhang
Microsoft Research Asia
Beijing, China
ygz@microsoft.com

ABSTRACT

Recently, Data Center Networking (DCN) has attracted many research attentions and innovative DCN designs have been proposed [1, 2]. All these designs need specialized packet forwarding engines due to their special routing algorithms, which are either based on commonly used packet headers or self-defined ones. Although programmable forwarding devices are available, it is difficult to use them to prototype these DCN designs, especially when self-defined headers are introduced. In this paper, we present a hardware based Configurable pAcket Forwarding Engine (CAFE) to facilitate the prototyping process. Through simple APIs, CAFE can be easily configured to forward self-defined packets, modify, insert, and delete arbitrary packet header fields without re-designing the hardware. We have implemented CAFE using NetFPGA. Evaluation demonstrates that CAFE can be easily configured and it can forward packets at line-rate.

Categories and Subject Descriptors

C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*ethernet, highspeed*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*routers*

General Terms

Design

Keywords

Configurable packet forwarding engine, data center networking, NetFPGA

*This work was performed when Yunfeng was an intern at Microsoft Research Asia.

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.
PRESTO'09, August 21, 2009, Barcelona, Spain.
Copyright 2009 ACM 978-1-60558-446-1/09/08 ...$5.00.

1. INTRODUCTION

Data centers with hundreds of thousands of servers are becoming important infrastructures for cloud computing. As a data center is always under complete control of a single organization, the underlying assumption for data center networking (DCN) becomes very different from that of the Internet. Recently, DCN has attracted many research attentions. Many innovative designs have been proposed to interconnect servers in data centers to build high capacity networks using commodity devices [1, 2, 3, 4, 5].

Currently there are mainly two approaches for DCN designs. One is to leverage servers for packet forwarding [2, 3, 5], and the other is to program the switches to build Clos networks such as fat-tree [1, 4]. Yet it is difficult to prototype these DCN designs due to two reasons. First, all these designs have special routing requirements, such as load-balancing and fault-tolerance. To meet these requirements, these designs use special forwarding schemes by either reusing existing headers or introducing new L2.5 headers. Hence, conventional packet forwarding devices such as Ethernet switches and IP routers cannot support these forwarding schemes without modification. Second, compared with the hardware approach, although software based forwarding devices can be easily programmed to support these forwarding schemes, currently they cannot achieve comparable high forwarding performance (with low CPU overhead), which is an important goal of these DCN designs.

A fully programmable hardware such as NetPFGA [6] becomes a good candidate for prototyping DCN designs. However, designing hardware on FPGA is time consuming and needs special skills, especially for achieving high performance. Besides, it is quite painful to debug designs on FPGA. To this end, a high performance hardware forwarding engine which can be easily configured to support self-defined packets will greatly facilitate DCN prototyping.

In this paper, we present a Configurable pAcket Forwarding Engine (CAFE) which provides simple yet powerful APIs to control its forwarding behavior. Compared with a fully programmable FPGA, it requires no hardware re-designing which greatly eases the development while guarantees high performance. Through simple APIs, CAFE allows users to define their own rules to classify packets into different types, and use different forwarding schemes for different packet types. This feature enables CAFE to forward packets of

multiple protocols simultaneously. Thus, CAFE can also virtualize the physical substrate to enable different protocols running on the same physical infrastructure [7].

We have implemented CAFE on top of NetFPGA. According to our experience, only tens or hundreds of lines of configuration scripts can configure CAFE into the forwarding device of a different DCN design. Experiments also show that all the ports of CAFE can forward packets at line-rate.

The rest of paper is organized as follows: In § 2, we describe the background of three DCN designs. We present the design and implementation of CAFE in § 3. We evaluate our NetFPGA-based CAFE in § 4 and present two concrete DCN examples built from CAFE. We discuss related work in § 5 and conclude in § 6.

2. BACKGROUND

In this section, we discuss the special packet forwarding mechanisms used by three DCN designs.

Fat-tree [1] is a DCN design in which multiple layers of switches are used to provide high network capacity. There are two types of switches: the core switches and the pod ones. The core switches are normal IP routers. The pod switches use specialized two-level lookup algorithm to forward IP packets based on their destination addresses. The 32-bit destination IP address is divided into two spaces. For the IP addresses assigned to that pod, the three most significant octets of the destination IP address are used as the lookup key. For other IP addresses, the least significant octet of the destination IP address is used as the lookup key.

DCell [2] is another DCN design which targets for very large data centers. In this design, every server is equipped with a multi-port NIC and needs to forward packets for other servers. To support its routing algorithm, DCell adds a header between the Ethernet and IP headers and servers forward packets based on the DCell header. DCell header is very similar to IP header. The main difference lies in that apart from a destination DCell Address, DCell header has a *Proxy DCell Address* and 1-bit Proxy Flag (PF) field. When PF is zero, the packet forwarding is based on the destination address. Otherwise, it is based on the proxy address. The proxy address is similar to the idea of IP loose source routing option.

BCube [5] is a DCN design targeting for modular data centers. BCube is similar to DCell in that servers are equipped with a multi-port NIC to forward packets for other servers. However, BCube uses source routing and designs a new layer 2.5 header to support the routing schema. The header has an array of Next Hop Address (NHA) fields and a Next Hop Index (NHI) field. The NHA fields store the addresses of all the nodes along the forwarding path. When an intermediate node receives a BCube packet, it uses the NHI field to locate the current NHA field and uses NHA to lookup the forwarding table. Then, the node increases NHI by one and sends the packet to the next hop.

3. DESIGN OF CAFE

We have three design goals for our forwarding engine:

- **Configurable forwarding behaviors**: As DCN designs have special routing algorithms, the underlying forwarding device should be configurable in order to support different forwarding schemes for both existing protocol headers and self-defined headers.

Device	Operations	Type
IPv4 router	Check *Version*, *Header Length* and *TTL* fields. Verify checksum.	1
	Forwarding table lookup based on the destination IP.	2
	Decrease *TTL* field. Update checksum.	3
Fat-tree switch	Check *Version*, *Header Length* and *TTL* fields. Verify checksum.	1
	Two-level forwarding table lookup based on the destination IP.	2
	Decrease *TTL* field. Update checksum.	3
DCell server	Check DCell *Version*, *Header Length* and *TTL* fields. Verify checksum.	1
	Forwarding table lookup based on either destination or proxy DCell address.	2
	Decrease *TTL* field. Update checksum.	3
BCube server	Check BCube *Version*, *Header Length* and *TTL* fields. Verify checksum.	1
	Forwarding table lookup based on the NHA field indexed by the NHI field.	2
	Decrease *TTL* field. Update checksum.	3
MPLS router	Check *TTL* field.	1
	Forwarding table lookup based on the MPLS label field	2
	Rewrite, push or pop MPLS label. Decrease *TTL*.	3

Table 1: All required operations in fast forwarding path for five devices.

- **Clear and simple interface to users**: A clear and simple interface can tell the exact capability of this hardware device and let users configure it easily.
- **Line-rate forwarding performance**: As DCN designs all aim at high capacity, the hardware forwarding device should provide high performance to make the prototype meaningful to researchers.

In the following sections, we first investigate the necessary operations in the fast forwarding path. Then we present our hardware design, and the software interface to control its forwarding behavior.

3.1 Operations in Fast Forwarding Path

Table 1 lists all the required operations in fast forward path (FFP) for five devices, i.e. IPv4 router, Fat-tree switch, DCell server, BCube server and MPLS router. In the table, column *type* denotes the type of operation listed below. As we only target for the most basic operations in packet forwarding, firewalls and Intrusion Detection Systems are not investigated. IPv4 router is the most widely deployed packet forwarding device and has the most basic operations along FFP. Fat-tree switch also forwards IP packets but uses a different forwarding table lookup algorithm based on the destination IP, i.e. two-level lookup algorithm (see § 2). DCell header is similar to IP header, but the forwarding table lookup can be based on different fields in the header. The uniqueness of forwarding table lookup of BCube header lies in that the position of the lookup key is not fixed, but is determined by another field in the header. MPLS router represents another type of switching technology, circuit switching. To sum up, there are following three types of operations along the FFP for these five devices.

1. Header verification. When a packet is received, the forwarding device checks its header to determine whether the packet is to be discarded, delivered to the slow path, or

forwarded. There are two types of verifications. The first type is header field verification. It compares the value of a certain field with a given value. For example, an IPv4 router checks whether TTL is zero, and whether the header length is 5. When TTL is zero, the packet is discarded. When the header length is not 5, it is delivered to slow path for further processing. To note, IPv4 header length is a 4-bit field.

The other type is checksum verification. It is different from the first type in that it uses 16-bit addition operations to sum up the whole packet header.

2. Forwarding table lookup. The device first builds a lookup key by extracting a set of fields from packet header, and then uses the key to locate an entry in the forwarding table. However, the device may use different lookup keys to forward packets. For example, the lookup key for a fat-tree switch can either be the three most significant octets or the least significant octet of destination IP. The lookup key for a DCell server can either be the destination address or proxy DCell address. Besides, the position of the lookup key can be determined by another field in the packet header, e.g. BCube packet forwarding.

3. Header modification, insertion and deletion. After the forwarding decision is made, the packet usually needs to be modified before sent. The operation can be field modification such as TTL decrement, checksum update, and MPLS label swapping. Or it can be header insertion and deletion such as MPLS label insertion and deletion when packets enter and exit the MPLS network.

As for the field modification, a new value can be used to overwrite the field such as swapping the MPLS label, or a new value is used to update the original value of the field such as decreasing TTL by one and updating checksum. As checksum protects the whole header, any changes in the header lead to its checksum update. For the commonly used Internet checksum, an adjusted value is added to the old checksum and the value only depends on the differences between the original and updated header [8]. Thus, the adjusted value can be decided when all the other header modification rules are known. Some modification operations are *type-specific*, which means they are the same for all the packets of a certain type, such as TTL decrement for IP packets. The others are *flow-specific* as they may differ from flow to flow. For example, MPLS label swapping is flow-specific. The new label depends on the old label.

The header insertion and deletion are flow-specific too. For example, an edge MPLS router gateway may have multiple internal MPLS links and external IP links. Only the flows across the two types of links need to perform these two operations. As for the content of the inserted header, some fields are flow-specific while others may vary packet-by-packet. For example, when an MPLS header is inserted into an IP packet, the 20-bit label, 3-bit experimental field and 1-bit stack field are flow-specific. On the other hand, the 8-bit TTL may vary packet-by-packet, as it is equal to IPv4 TTL minus one.

3.2 Hardware Design

Based on the analysis above, a desired forwarding engine for DCN must meet the following requirements:

- **Bit-level field verification**. The engine must allow user to select any bits from a header and compare them with the given data.

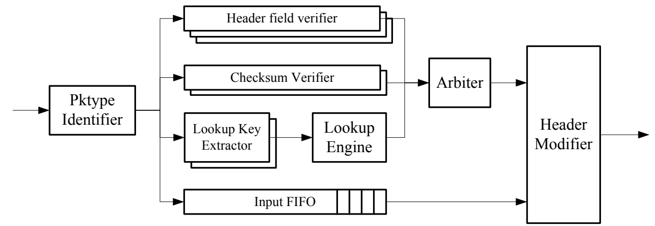

Figure 1: Block diagram of CAFE

- **Bit-level lookup key extractor**. The engine must allow user to extract any combination of bits from a header as the lookup key.

- **Bit-level field modification**. The engine must allow user to select any bits to be modified by a user-supplied value. Addition and replacement must be provided to support field updating and overwriting. The engine must support both type-specific and flow-specific modification operations.

- **Byte-level header insertion and deletion**. The engine must allow user to insert a multi-byte data into the header, or to delete a multi-byte data from the header. As for the insertion, its position, length and data must be flow-specific and the engine must be able to apply bit-level field modification to the inserted data. As for the deletion, its position and length must be flow-specific too.

- **Type-based operation**. As packets may need different forwarding processes, the engine must be able to classify packets into different types and apply different operations to each type.

Based on these requirements, we designed CAFE as shown in Figure 1. CAFE processes packets in three phases. When a packet arrives, *Pktype Identifier* module first identifies the type of the packet. When its type cannot be identified, it means the packet cannot be forwarded by hardware, and the packet will be delivered to the slow path.

In the second phase, CAFE pushes the packet data into its input FIFO, and sends its copy to do header verification and forwarding table lookup at the same time. There are multiple *header field verifiers*, *lookup key extractors* and *checksum verifiers*. Based on the packet type, one key extractor, one checksum verifier and several header field verifiers are selected to perform header verification and table lookup operations.

A *header field verifier* filters the packets which should not be forwarded by hardware. The *checksum verifier* verifies the packet's checksum. The *lookup key extractor* extracts a set of packet fields as the lookup key, and the *Lookup Engine* looks up the forwarding table using this key. Every entry in the forwarding table represents a flow. The entry contains the flow-specific modification and insertion data of that flow.

The *arbiter* checks the results of these selected operations. If the packet is filtered by a header field verifier, has wrong checksum or fails to match to a forwarding table entry, the packet is delivered to the slow path or discarded without modification. Otherwise, the *header modifier* module will further modify the packet before sending it out.

In the third phase, the packet is associated with not only a packet type but also a flow entry. So the *header modifier* can use both the type-specific data and flow-specific data to modify the packet header.

3.2.1 Packet identifier and Header field verifier

The packet identifier consists of a number of parallel filters. When a packet matches a filter, it is marked as the packet type associated with the filter. When it matches more than one filters, the filter with the highest priority overrides. When it does not match any filter, its type cannot be identified and the packet will be delivered to the slow path. The header field verifier is also a filter. When a packet matches this filter, the packet should not be forwarded by hardware.

A filter performs multiple independent comparisons between packet header fields and the corresponding user-supplied data. If all the comparisons of a filter return true, the packet matches the filter. A header field is selected by a direct *Field Seizer* (FS). User can configure a direct FS with a field address and a 1-byte mask. The seizer directly extracts a 1-byte field at that address and uses the mask to select the needed bits within that byte. The field address is counted from the beginning of Ethernet header to that field in unit of byte.

The power of the filter is that the seizers are all independent and can extract fields in different positions. Thus they can extract arbitrary combination of bits in the packet header, such as 16-bit ethertype plus 32-bit IP address to filter packets towards a certain IP address. Obviously, the number of seizers in each filter determines the CAFE's matching capability.

3.2.2 Lookup Key Extractor and Lookup Engine

A *lookup key extractor* consists of several direct FSs and indirect FSs. The FS is described in previous section. The indirect FS is used for source routing. It has three parameters: two addresses and one mask. It first selects 1-byte data from the 1st address as an offset, and then sums the offset and the 2nd address to obtain a new byte position. Finally, it selects 1-byte data at the new position and uses the mask to get the final data. Each FS can be independently enabled and disabled. When it is disabled, it outputs zero. The lookup key consists of the packet type and outputs of all the seizers. The packet type is used to differentiate lookup keys for different packet types so that they can share the same forwarding table.

The *Lookup Engine* has a forwarding table. It locates an entry in the forwarding table according to the key. Each entry represents a flow. It contains the lookup key, per-flow modification data and insertion data, and the index to the neighbor table. The hardware forwarding table is maintained by the software. When a packet does not match any entry, it is delivered to the slow path. Software will perform the forwarding and table updating, so that the subsequent packets can then be processed by the hardware. The neighbor table has two columns, the output port number and next hop MAC address. The first one denotes the output port of the packet and the second one is used to replace the destination MAC address of the packet. The neighbor table is also maintained by software.

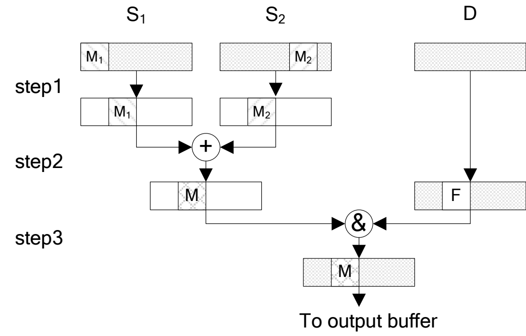

Figure 2: 64-bit field modification engine

3.2.3 Header Modifier

The header modifier first uses a selector to perform header insertion and deletion. The selector has two inputs and one output. The two inputs are the original packet data from the input FIFO and the insert data from a flow entry. The output is the field modification engine. Normally, the selector pops data from the input FIFO and pushes it to the output. To achieve insertion, the selector suspends popping and pushes insert data to the output. To achieve deletion, the selector pops data from the input FIFO and simply drops it.

The field modification engine shown in Figure 2 achieves bit-level field modification. It is designed for 64-bit wide data bus. All three inputs and the output are 64-bit wide. S_1 is supplied by user. It is from either type-specific registers or modification data in the flow entry. S_2 is seized from original packet. D is from the output of the selector. M_1 and M_2 are two fields used to replace field F in D. In step 1, M_1 and M_2 are shifted to be aligned with F and a *mask* is applied to them to zero all other bits in S_1 and S_2. In step 2, we sum M_1 and M_2. Depending on the configuration, the carry bit is either cleared or added back into the resulting sum. The latter operation is used in updating Internet checksum. Meanwhile, F is zeroed. In step 3, M replaces F by OR operation. To note, when M_1 or M_2 is zero, M is the same as M_2 or M_1.

This engine can perform addition and replacement operations with two sources and one destination. It can be used to decrease TTL, update checksum, rewrite MPLS label and etc. For example, to decrease IPv4 TTL by one, we can set M_1 to 0xFF, set both M_2 and F to the original TTL field, and set the *mask* and two *shift*s accordingly.

3.3 Software API

It is difficult to directly use over 300 registers provided by the hardware to configure its packet forwarding behaviors. As several registers need to be configured cooperatively for one behavior, e.g. building the forwarding table lookup key, we use one software API to configure all registers related to one function. These software APIs let users easily configure every functional module in CAFE, such as the filter in packet type identifier, the lookup key extractor, and etc. APIs for writing forwarding table are also provided. Altogether, there are 13 core APIs plus a few auxiliary APIs. Due to space limitation, we can only show some of these APIs in an example in the next section.

4. IMPLEMENTATION AND EVALUATION

We've implemented CAFE using NetFPGA. Due to its resources limitation, CAFE supports two different user-defined packet types in current implementation. For each packet type, there are one lookup key extractor, one checksum verifier and two filters as header field verifiers. Each filter has eight direct FSs, which are enough for our experiments. Also due to limited FPGA resources, the lookup key is set to be 128-bit. It consists of 1-byte packet type, and 15-byte seized data from 11 direct FSs and 4 indirect FSs. For each flow entry, we have 16 bytes modification data and a maximum of 24 bytes insertion data.

The forwarding table is implemented using the combination of a hash table of 1K entries and a TCAM table of 32 entries. As NetFPGA can not support a large TCAM, the hash table is taken as the main storage for the flow entries, while the TCAM table is used to resolve hash conflicts.

CAFE uses 60% LUTs, 44% BRAMs of Virtex2Pro50 FPGA. We also downloaded the source code of openflow 0.8.9-1 and synthesised it. It (excluding I/O modules) uses about 40% LUTs, 24% BRAMs of Virtex2Pro50 FPGA. Thus, to provide the flexibility to forward self-defined packets and its header modification function, CAFE only uses an extra 20% LUTs and 20% BRAMs of the same FPGA than the Openflow.

4.1 Experimental setup

We use a NetFPGA packet generator to connect all the four Gigabit ports of CAFE. The generator are used to generate packets at wire speed and check the output packets forwarded by CAFE. We have done two experiments on this testbed. First we configure CAFE as a fat-tree switch, and then configure it to forward DCell packets.

Fat-tree. In this experiment, we configure CAFE as a fat-tree pod switch. The pod switch uses /24 prefix to look up the IP space assigned to the pod, and it uses /8 suffix to look up the rest of IP addresses. Thus for CAFE, we simply treat these two spaces as two packet types and build different lookup keys for them.

Figure 3 shows the script to configure pod switch 10.2.2.1 (see Figure 3 in [1]). In all these APIs, the first argument denotes the packet type index.

Line 1-4 define that IPv4 packets whose destination address is 10.2. $*$.$*$ as type 0. Line 5-6 define the rest IPv4 packets as type 1. **set_pktype** configures a comparison of a filter for a packet type. The args are packet type index, filter index, seizer index, the address of the 1-byte field, its mask, the user supplied value and a flag. A filter with smaller index has higher priority. The address is counted from the beginning of Ethernet header to this field in unit of byte. The comparison result returns true when flag is 0 and two inputs are equal, or when flag is 1 and two inputs are not equal.

Line 7-14 perform IP header verification for type 0 and 1. Line 7-8 verifies the IPv4 version, header length and TTL fields. Line 9-10 says that when the packet is to 10.2.2.1, deliver the packet to slow path. Line 11 verifies its IP checksum. **set_filter** configures a comparison of a header field verifier. The args are exactly the same as **set_pktype**. **set_csum_verify** verifies the Internet checksum of a data block. The args are packet type index, the starting address of the block and the length of the block.

Line 15-17 configure the first three destination IP octets

```
 # configure packet type identifier
 1: set_pktype(0, 0, 0, 13, 0xFF, 0x08, 0) // ethertype
 2: set_pktype(0, 0, 1, 14, 0xFF, 0x00, 0) // ethertype
 3: set_pktype(0, 0, 2, 31, 0xFF, 0x0A, 0) // dst ip
 4: set_pktype(0, 0, 3, 32, 0xFF, 0x02, 0) // dst ip
 5: set_pktype(1, 1, 0, 13, 0xFF, 0x08, 0) // ethertype
 6: set_pktype(1, 1, 1, 14, 0xFF, 0x00, 0) // ethertype
 # header verification
 7: set_filter(0, 0, 0, 15, 0xFF, 0x45, 1) // ver & hdrlen
 8: set_filter(0, 1, 0, 23, 0xFF, 0x00, 0) // ttl
 9: set_filter(0, 2, 0, 33, 0xFF, 0x02, 0) // dst ip
10: set_filter(0, 2, 1, 34, 0xFF, 0x01, 0) // dst ip
11: set_csum_verify(0, 15, 20) // ip header checksum
12: set_filter(1, 0, 0, 15, 0xFF, 0x45, 1) // ver & hdrlen
13: set_filter(1, 1, 0, 23, 0xFF, 0x00, 0) // ttl
14: set_csum_verify(1, 15, 20) // ip header checksum
 # build lookup key extractor
15: set_direct_key(0, 0, 31, 0xFF) // 1st octet of dst ip
16: set_direct_key(0, 1, 32, 0xFF) // 2nd octet of dst ip
17: set_direct_key(0, 2, 33, 0xFF) // 3nd octet of dst ip
18: set_direct_key(1, 0, 34, 0xFF) // 4th octet of dst ip
 # decrease TTL, update checksum
19: set_type_modify_data(0, 0xFF01000000000000)
20: set_type_modify_rule(0, 0, 3, 0x000000000000FF00,
                         3, 6, 0, 1)
21: set_type_modify_rule(0, 1, 4, 0xFFFF000000000000,
                         4, -1, 0, 2)
22: set_type_modify_data(1, 0xFF01000000000000)
23: set_type_modify_rule(1, 0, 3, 0x000000000000FF00,
                         3, 6, 0, 1)
24: set_type_modify_rule(1, 1, 4, 0xFFFF000000000000,
                         4, -1, 0, 2)
 # set hash table
25: set_hash_table(393, 0x000A0200 00000000
                   00000000 00000000, 0x0, 0x0)
26: set_hash_table(139, 0x000A0201 00000000
                   00000000 00000000, 0x0, 0x1)
27: set_hash_table(323, 0x01020000 00000000
                   00000000 00000000, 0x0, 0x2)
28: set_hash_table(386, 0x01030000 00000000
                   00000000 00000000, 0x0, 0x3)
 # set neighbor table
29: set_nb_table(0, 0x0, 0x0018FE2ED6EA)
30: set_nb_table(1, 0x1, 0x0018FE2E046E)
31: set_nb_table(2, 0x2, 0x0018FE2ED24A)
32: set_nb_table(3, 0x3, 0x0018FE2E00F2)
```

Figure 3: Configuration script for a fat-tree pod switch

as the lookup key for packet type 0. Line 18 configures the last destination IP octet as the lookup key for packet type 1. **set_direct_key** configures a direct FS of the lookup key extractor for a packet type. The args are packet type index, seizer index, the address of the 1-byte field and its mask.

Line 19-24 decrease TTL by one and update the checksum accordingly. **set_type_modify_data** configures S_1 in the field modify engine. The args are packet type index and S_1. In this case, S_1 is used to modify TTL and checksum. The 1st byte of S_1 (0xFF) is used to decrease TTL. The 2nd and 3rd bytes of S_1 (0x0100) are used to update the checksum. **set_type_modify_rule** configures other parameters of the field modify engine. The args are packet type index, rule index, 64-bit block count of S_2, mask, 64-bit block count of D, shift for S_1, shift for S_2 and a flag. The block counts are which 64-bit blocks of S_2 and D locate. The shift is the number of bytes to shift. Negative number means the direction is from right to left. As the source field M_2 and the target field F are the same for TTL decrement and checksum update, the shift for S_2 is zero. When the flag is 1, the carry

bit is cleared. When it is 2, the carry bit is added back in.

Line 25-28 configure the forwarding lookup table. The forwarding table is configured according to Figure 4 in [1]. **set_hash_table** sets a forwarding table entry. The args are the index of the entry, the 128-bit lookup key, the per-flow modification data and the neighbor table index. The hash function is CRC-16. As the hash table has only 1024 entries, only last 10 bits of the hash value is used as the index. In this case, as there is no per-flow modification data, the 3rd arg is set to zero.

Line 29-32 configure the neighbor table. **set_nb_table** sets a neighbor table entry. The args are the entry index, the output port and the next hop MAC address.

DCell. In this experiment, we configure CAFE to forward packets as a DCell server. We classify packets according to their PF flag. When it is zero, it uses the destination DCN address as the lookup key. Otherwise, it uses proxy DCN address as the lookup key. Other configurations such as header verification, TTL and checksum modifications are similar to those of fat-tree.

4.2 Forwarding Performance

The packet generator evenly sends traffic from every port to all other three ports at wire speed. Table 2 shows that CAFE is capable of forwarding packets almost at the line-rate across 64, 512, 1024, and 1514 packet sizes. The performance of CAFE to act as fat-tree and DCell switch are the same.

Pkt Size (Bytes)	64	512	1024	1514
Fwd Rate (Gbps)	4.0	3.8	3.8	3.8

Table 2: Forwarding rate of NetFPGA-based CAFE

5. RELATED WORK

Openflow proposes a standard interface to manipulate the forwarding table of a proprietary switch. Openflow type-0 can only handle standard TCP/UDP + IP + Ethernet packets and cannot fully meet the requirements of DCN. For example, Openflow type-0 cannot meet the requirement of DCell in which new DCell header is introduced. And it cannot be applied in BCube and Monsoon where source routing is used. Our work may be considered to provide a concrete example on what kind of functionalities are needed in Openflow type-1.

Casado et al. propose a new flow-oriented approach for packet forwarding hardware [9], in which the forwarding decision is done in software and the hardware just mimics what the software tells it. CAFE can be used to implement their design. Its generic packet field extractor can be used to match a packet to a forwarding table. CAFE also addresses other important issues such as header modification, insertion and deletion in the hardware forwarding path.

Orphal [10] defines a set of APIs to control forwarding behavior on proprietary switches or routers. The APIs are mainly limited to TCAM, which provides similar functionalities as CAFE's lookup engine. CAFE focuses more on flexible header extraction and modification which are not addressed in Orphal but are important for non-IP packet forwarding in DCN.

Many packet processing engines [11, 12] are available to process complex network protocols such as IPSec and TCP, or to perform deep packet inspections, such as application classification and signature matching. Our design mainly targets for the common operations in packet forwarding, which are much more simpler than those protocol states and packet content analysis.

6. CONCLUSION AND FUTURE WORK

CAFE is a configurable packet forwarding engine which provides simple APIs to control its packet forwarding behavior. Due to its configurability, packet header verification, forwarding table lookup, header modification, insertion and deletion can all be performed without understanding the packet header semantics. Our initial experimental experience with CAFE is promising as we can now easily re-configure CAFE to forward self-defined packet headers without any hardware re-designing.

CAFE is still in active development. Current header insertion and deletion design can only achieve 8-byte level operation. We are working towards a genuine byte-level design. As one might expect, there are limitations on the types of operations that CAFE can provide. For example, CAFE right now only supports Internet checksum verification, and packet modification rule is limited to simple replacement and addition with limited inputs. As part of our future work, we are trying to make CAFE a generic and high performance hardware forwarding engine for DCN research.

7. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proc. SIGCOMM*, 2008.

[2] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: A Scalable and Fault Tolerant Network Structure for Data Centers," in *SIGCOMM*, 2008.

[3] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, and S. Lu, "FiConn: Using Backup Port for Server Interconnection in Data Centers," in *IEEE INFOCOM*, 2009.

[4] A. Greenberg, D. Maltz, P. Patel, S. Sengupta, and P. Lahiri, "Towards a Next Generation Data Center Architecture: Scalability and Commoditization," in *SIGCOMM PRESTO Workship*, 2008.

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

[6] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "NetFPGA: Reusable Router Architecture for Experimental Research," in *PRESTO*, 2008.

[7] T. Anderson, L. Peterson, S. Shenker, and J. Turne, "Overcoming the Internet Impasse Through Virtualization," in *ACM HOTNETS III*, 2004.

[8] T. Mallory and A. Kullberg, "Incremental updating of the Internet checksum," 1990. RFC1141.

[9] M. Casado, T. Koponen, D. Moon, and S. Shenker, "Rethinking Packet Forwarding Hardware," in *ACM HotNets-VII*, 2008.

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

[11] Netronome, "Network Flow Processor NFP-3200 Product Brief." `http://www.netronome.com/files/file/Netronome%20NFP%20Product%20Brief%20(3-09).pdf`.

[12] C. L. Hayes and Y. Luo, "Dpico: a high speed deep packet inspection engine using compact finite automata," in *ANCS*, (New York, NY, USA), pp. 195–203, ACM, 2008.