Implementing an OpenFlow Switch on the NetFPGA platform

Jad Naous
Stanford University
California, USA
jnaous@stanford.edu

David Erickson
Stanford University
California, USA
derickso@stanford.edu

G. Adam Covington
Stanford University
California, USA
gcoving@stanford.edu

Guido Appenzeller
Stanford University
California, USA
appenz@cs.stanford.edu

Nick McKeown
Stanford University
California, USA
nickm@stanford.edu

ABSTRACT

We describe the implementation of an OpenFlow Switch on the NetFPGA platform. OpenFlow is a way to deploy experimental or new protocols in networks that carry production traffic. An OpenFlow network consists of simple flow-based switches in the datapath, with a remote controller to manage several switches. In practice, OpenFlow is most often added as a feature to an existing Ethernet switch, IPv4 router or wireless access point. An OpenFlow-enabled device has an internal flow-table and a standardized interface to add and remove flow entries remotely.

Our implementation of OpenFlow on the NetFPGA is one of several reference implementations we have implemented on different platforms. Our simple OpenFlow implementation is capable of running at line-rate and handling all the traffic that is going through the Stanford Electrical Engineering and Computer Science building. We compare our implementation's complexity to a basic IPv4 router implementation and a basic Ethernet learning switch implementation. We describe the OpenFlow deployment into the Stanford campus and the Internet2 backbone.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks; C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Design, Management, Performance

ANCS '08, November 6–7, 2008, San Jose, CA, USA.

Keywords

Computer networks, Flow switching, NetFPGA, OpenFlow, Packet switching, Programmable networks

1. INTRODUCTION

Today it has become extremely difficult to innovate in the computer networks that we use everyday in our schools, businesses, and homes. Current implementations of mainstream network devices, such as Ethernet switches and IP routers, are typically closed platforms that can not be easily modified or extended. The processing and routing of packets is restricted to the functionality supported by the vendor. And even if it were possible to reprogram network devices, network administrators would be loathe to allow researchers and developers to disrupt their production network. We believe that because network-users, network-owners and network-operators can not easily add new functionality into their network, the rate of innovation—and improvement—is much lower than it could be. Rather than leave network innovation to a relatively small number of equipment vendors, our goal in the OpenFlow project [19] is to enable a larger population of researchers and developers to evolve our networks, and so accelerate the deployment of improvements, and create a marketplace for ideas. Today, most new ideas proposed by the research community—however good—never make it past being described in a paper or conference. We hope that OpenFlow will go so some way to reduce the number of lost opportunities.

OpenFlow is described in more detail in [12], along with a number of motivating applications.

In brief, an OpenFlow network consists of OpenFlow compliant switches and OpenFlow controller(s) with *unmodified* end-hosts. Essentially, OpenFlow separates the "datapath" over which packets flow, from the "control path" that manages the datapath elements. The datapath elements are flow-switches, consisting of a flow-table and the means to talk to a remote Controller using the OpenFlow Protocol. A flow is defined as all the packets matching a flow-entry in a switch's flow-table. Flow entries are quite general, and resemble ACL entries found in firewalls—although here they can include fields from layers 2, 3 and 4. The controller decides which flows to admit and the path their packets should follow. To some extent, OpenFlow resembles previous attempts to separate control from the datapath: for example,

IP Switching [13] and 4D [5]. OpenFlow differs in that it only defines the flow-based datapath switches, and the protocol for adding and deleting flow entries. It deliberately does not define the controller—the owner of an OpenFlow network is free to use any controller that speaks the OpenFlow protocol—it could be written by a researcher, downloaded from a repository, written by the network owner, or any combination. Available controller include the policy based controller NOX [17][1] as well as a testing controller included with the OpenFlow test suite. We expect and encourage other controllers to follow.

The simplest OpenFlow switch consists mainly of a flow table and an interface for modifying flow table entries. Each entry consists of a flow description and an action associated with that flow. In practice, OpenFlow is usually added as a feature to an existing Ethernet switch, router or access point. Experimental traffic (usually distinguished by VLAN ID) is processed by the flow-table, while production traffic is processed using the normal Layer-2 or Layer-3 protocols. However, in this paper, we restrict our interest to a "pure" OpenFlow switch that implements the OpenFLow Type 0 specification [18].

The OpenFlow controller establishes a Secure Socket Layer (SSL) connection to an OpenFlow switch and uses the OpenFlow interface to add, remove, and update flow table entries. If a packet arrives to a switch and it is not matched in the flow table, it is encapsulated and sent to the controller over the SSL channel. The controller can then examine it, update any flow table entries in the network, and send the packet back to the switch.

We are working with several vendors to add OpenFlow as a feature to their products, either as a supported feature, or on an experimental basis. For example, working with HP Labs and Cisco researchers, OpenFlow has been added—on an experimental basis—to the ProCurve 5400 and Catalyst 6500 series switches respectively. NEC has added OpenFlow to the IP8800 router, and we are nearing completion of a prototype on top of the Junos SDK in the Juniper MX-series router. We expect several other announcements in the coming months. We are deploying OpenFlow as the production network in the Gates Computer Science and the Center for Integrated Systems (CIS) buildings at Stanford; and on 100 WiFi access points to which OpenFlow has been added. OpenFlow is also being deployed in the Internet2 backbone [4] on our experimental NetFPGA network.

This paper describes the implementation of a full line-rate 4-port 1-GigE OpenFlow switch on NetFPGA [16]. NetFPGA is a platform that enables researchers and students to experiment with Gigabit rate networking hardware. It consists of a PCI card that has an FPGA, memory (SRAM and DRAM), and four 1-GigE Ethernet ports. Hardware description source code (gateware) and software source code is freely available online for building sample designs such as a 4-port IPv4 router and a 4-port NIC.

We will describe the implementation of OpenFlow on the NetFPGA and compare its complexity to other standard packet switching elements—an Ethernet switch and an IPv4 router. We will also compare it to a standard 4-port NIC implementation. Section 2 describes OpenFlow in more detail, section 3 describes NetFPGA, section 4 describes the OpenFlow implementation on the NetFPGA, section 5 discusses our implementation results, section 6 describes our OpenFlow deployment at Stanford and in Internet2, section 7 describes works that are similar or related to OpenFlow and NetFPGA, and section 8 concludes the paper.

[1]OpenFlow and NOX both share a common heritage from Ethane [1].

2. OPENFLOW ARCHITECTURE

OpenFlow is simple. It pushes complexity to controller software so that the controller administrator has full control over it. This is done by pushing forwarding decisions to a "logically" centralized controller, and allowing the controller to add and remove forwarding entries in OpenFlow switches. This places all complexity in one place where it can be managed, transferring the cost from every switch in the network to a single location.

Centralizing complexity allows the network administrator to keep close watch over the behavior of the network. Since she has tight and direct control over forwarding in the switches, she can manage network resources and separate production traffic from experimental traffic. The central controller can be programmed to behave as a multiplexer that splits traffic belonging to different network users onto different user-controlled OpenFlow controllers, all under the network administrator's control. This form of network virtualization allows researchers to run their own protocols on the physical network while leaving control in the hands of the network administrator. See NOX [6] for a powerful OpenFlow controller implementation.

The basic OpenFlow switch version "type", the OpenFlow Type-0 switch, classifies packets into flows based on a 10-tuple which can be matched exactly or using *wildcards* for fields. The following fields constitute the 10-tuple:

- Switch input port
- Source MAC address
- Destination MAC address
- Ethernet Type
- VLAN ID
- IP source address
- IP destination address
- IP protocol
- TCP/UDP source port
- TCP/UDP destination port

Flow table entries are matched using this 10-tuple to find the corresponding actions associated with the flow. The OpenFlow Type-0 switch has three required actions:

- Forward to a specified set of output ports: This is used to move the packet across the network.
- Encapsulate and send to the controller: The packet is sent via the secure channel to the remote OpenFlow controller. This is typically used for the first packet of a flow to establish a path in the network.
- Drop: Can be used for security, to curb denial of service attacks, or to reduce spurious broadcast discovery traffic from end-hosts.

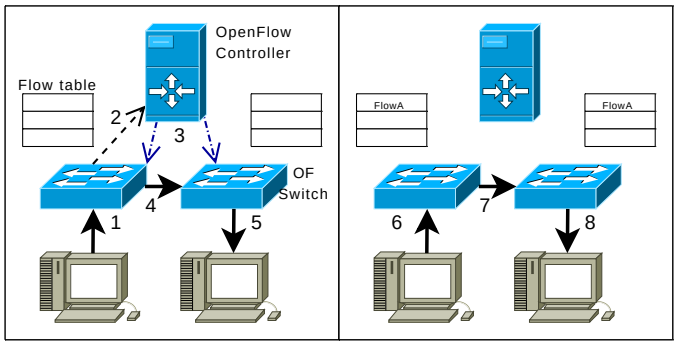

Figure 1: Steps when a new flow arrives at an OpenFlow switch.

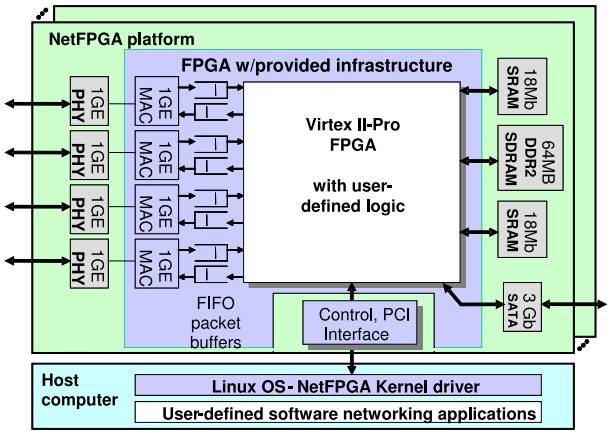

Figure 2: Detailed block diagram of the components of the NetFPGA board.

The protocol also specifies other optional per-packet modifications such as VLAN modification, IP address rewriting, and TCP/UDP port rewriting. Future revisions of the protocol describing more advanced "types" of OpenFlow switches will provide more general matching and actions.

If a match is not found for an arriving packet, the packet is sent to the controller which decides on the action(s) that should be associated with all packets from the same flow. The decision is then sent to the switch and cached as an entry in the switch's flow table. The next arriving packet that belongs to the same flow is then forwarded at line-rate through the switch without consulting the controller.

Figure 1 shows the steps for routing a flow between two hosts across two switches. In the diagram on the left, the switch flow tables are empty. When a new packet arrives in step 1, it is forwarded to the controller in step 2. The controller examines the packet and inserts entries into the flow tables of the switches on the flow's path in step 3. The packet is then sent through to the receiving host in steps 4 and 5. In steps 6,7, and 8 any new packets belonging to the same flow are routed directly since they would match the new entry in the flow tables.

More details on the protocol can be found in the OpenFlow whitepaper [12] and the protocol specification [18]. Readers familiar with Ethane [1] will notice OpenFlow's resemblance to Ethane's datapath. Indeed, OpenFlow *is* based on Ethane's datapath. It formalizes the datapath and provides an abstraction that allows users to build and extend beyond Ethane's scope. We are using OpenFlow in several projects here at Stanford such as wireless mobility, power-aware networking, and network virtualization.

3. NETFPGA

The NetFPGA platform consists of three parts: hardware, gateware, and software. The hardware is a PCI card that has the following core components:

- Xilinx Virtex-II Pro 50
- 4x 1 Gbps Ethernet ports using a soft MAC core
- Two parallel banks of 18 MBit Zero-bus turnaround (ZBT) SRAM
- 64 MBytes DDR DRAM

Figure 2 shows a more detailed block diagram of the components of the NetFPGA board. Several of these parts are donated by manufacturers for academic use, reducing the cost[2].

Software and gateware (Verilog HDL source code) are available for download under an open source license from netfpga.org. This allows jumpstarting prototypes and quickly building on existing designs such as an IPv4 router or a NIC. The gateware is designed to be modular and easily extensible. Designs are implemented as modular stages connected together in a pipeline, allowing the addition of new stages with relatively small effort [15]. The pipeline is depicted in Figure 3.

Gateware: There are two reference designs distributed with the official NetFPGA release that run on the NetFPGA: an IPv4 router, and a 4-port NIC. All reference designs are based on a generic reference pipeline shown in Figure 4. The reference pipeline captures the main stages of packet processing in a network switch. The Rx Queues pull packets from the I/O ports, the Input Arbiter selects which Rx Queue to service, the Output Port Lookup decides which output queue to store a packet in, the Output Queues module stores packets until the output port is ready, and the Tx Queues send packets out on the I/O ports.

The main switching decision usually happens in the Output Port Lookup stage and it differentiates an IPv4 router from a NIC or an Ethernet switch. We have also implemented a learning Ethernet switch which we will use for comparison in this paper.

Software: The software includes the NetFPGA device drivers, utilities, and two router controller packages that can populate the IPv4 Router's hardware forwarding table. The first is a stand-alone routing software package based on PW-OSPF [23] that runs entirely in user space. The second is a daemon that mirrors Linux's routing tables from memory into the hardware. This allows using standard open-source routing tools to easily build a full line-rate 4Gbps router using hardware acceleration.

4. OPENFLOW SWITCH

We have implemented an OpenFlow switch using NetFPGA to understand the limitations and difficulties faced by

[2]At the time of writing, boards are available for $500 for research and teaching.

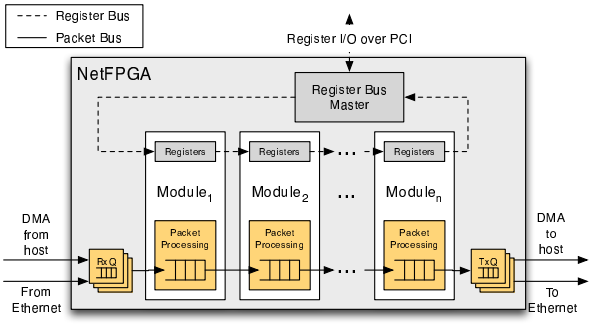

Figure 3: Modular NetFPGA pipeline structure: There are two main buses that traverse logic — the packet bus and the register bus. The high-bandwidth packet bus is used for packet processing while the register bus is used to carry status and control information between hardware modules and the software.

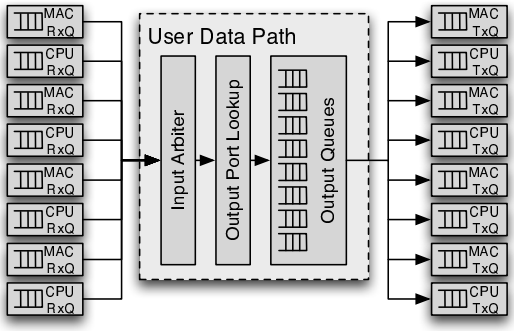

Figure 4: The IPv4 Router is built using the "Reference Pipeline" - a simple canonical five stage pipeline that can be applied to a variety of networking hardware.

OpenFlow switch designers. Our implementation can hold more than 32,000 exact-match flow entries and is capable of running at line-rate across the four NetFPGA ports. By using the currently reserved memory area for future expansion, the exact-match flow table can be expanded to hold more than 65000 entries.

4.1 Software

The OpenFlow switch management software extends the OpenFlow reference software implementation. This is an open-source software package for Linux that implements an OpenFlow switch in software and is available for download from the OpenFlow website [19]. The reference software can be divided into two sections: user-space and kernel-space.

The user-space process communicates with the OpenFlow controller using SSL to encrypt the communication. The OpenFlow protocol specifies the format for messages between the switch and controller. Messages from the switch to the controller such as arrival of new flows or link state updates and messages from the controller to the switch such as requests to add or delete flow table entries are exchanged between this user-space process and the kernel module via IOCTL system calls.

The kernel module is responsible for maintaining the flow tables, processing packets, and updating statistics. By default the reference OpenFlow switch kernel module creates these tables only in software, matching packets received via NICs on the host PC. The tables are linked together as a chain, and each packet is tested for a match sequentially in each table in the chain. Priority is given to the first table that matches in the chain. The wildcard lookup table in the software is implemented using a linear search table, while the exact lookup table is a hash table using two-way hashing.

The OpenFlow kernel module enables extensibility by allowing a secondary hardware-specific module to register additional tables with it. The additional flow tables take priority over the primary module's tables for all flow requests (including statistics reporting). We have extended the reference system by adding a NetFPGA OpenFlow kernel module. This module takes advantage of the OpenFlow kernel module interface and links the hardware tables with the primary OpenFlow module.

Packets arriving at the NetFPGA that match an existing entry are forwarded in hardware at line-rate. Packets that do not match an existing flow in hardware, i.e. new flows, are sent up to the OpenFlow kernel module which will then handle the packet, potentially forwarding it on to the controller.

In the event that the NetFPGA hardware flow tables are full, the kernel module will refuse to insert the entry into its flow table. This causes the entry to be inserted into the primary OpenFlow module's software flow tables. Future packets from such flows will not match in hardware, and will be passed up to software for processing.

4.2 Hardware

The OpenFlow implementation on the NetFPGA is shown in Figure 5. It uses the reference pipeline shown in Figure 4. Whereas, in the IPv4 router, the Output Port Lookup stage executes the longest prefix match (LPM) and ARP lookups, the OpenFlow Output Port Lookup stage does matching using the 10-tuple described above. The OpenFlow Lookup stage implements the flow table using a combination of on-chip TCAMs and off-chip SRAM to support a large number of flow entries and allow matching on wildcards.

As a packet enters the stage, the *Header Parser* pulls the relevant fields from the packet and concatenates them. This forms the flow header which is then passed to the *Wildcard Lookup* and *Exact Lookup* modules. The Exact Lookup module uses two hashing functions on the flow header to index into the SRAM and reduce collisions. In parallel with the Exact Lookup, the Wildcard Lookup module performs the lookup in the TCAMs to check for any matches on flow entries with wildcards. The TCAMs are implemented as 8 parallel 32-entry 32-bit TCAMs using Xilinx SRL16e primitives. These were generated using Xilinx's IP core generator—*coregen*— as in [25], [24].

The Exact Lookup is a state machine that is tightly synchronized with the SRAM controller. The internal pipeline runs at 125MHz with a 64-bit bus width, which means that it can handle 8 Gbps. Since minimum size packet is 8 words, then in the worst case we can expect a new packet every 16 cycles, and we need to perform one lookup every 16 cycles to maintain line-rate. The implemented state machine has 32 cycles, interleaving lookups from two packets. In the first

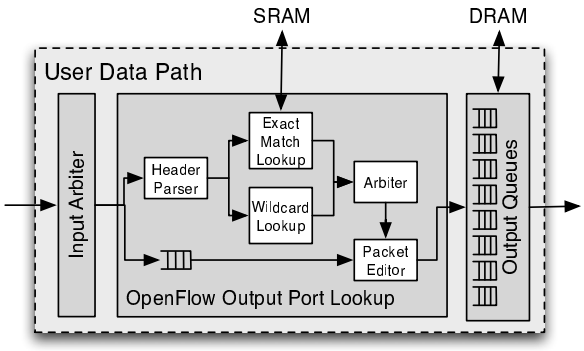

Figure 5: The OpenFlow switch pipeline is similar to the IPv4 Reference Router Pipeline.

8 cycles, the state machine reads the flow headers stored in the two locations indicated by the two hashes for the first packet. While waiting for the results of the lookups for the first packet, the state machine issues read requests for the flow headers using the second packet's two hashes. In the meantime, the results from the first packet's read requests are checked for a match. In the case of a hit, the data from the hit address is read. The same is then done for the second packet.

The results of both wildcard and exact-match lookups are sent to an arbiter that decides which result to choose. Once a decision is reached on the actions to take on a packet, the counters for that flow entry are updated and the actions are specified in new headers prepended at the beginning of the packet by the *Packet Editor*.

The design allows more stages—*OpenFlow Action* stages—to be added between the Output Port Lookup and the Output Queues. These stages can handle optional packet modifications as specified by the actions in the newly appended headers. It is possible to have multiple Action stages in series each doing one of the actions from the flow entry. This allows adding more actions very easily as the specification matures—the current implementation does not support *any* packet modifications.

5. RESULTS

In this section we will first give performance results for our OpenFlow switch. We will then compare the complexity of the Output Port Lookup module for the OpenFlow Switch, the IPv4 Router, the NIC, and the learning Ethernet switch.

5.1 Performance Results

Our NetFPGA OpenFlow switch implementation is evaluated in three dimensions: flow table size, forwarding rate, and new flow insertion rate.

Table Size: While modern enterprise routers and switches can have tables that are hundreds of thousands of entries (our Gates building router—a Cisco Catalyst 6509—can fit 1M prefixes), the total number of active flows at any point in time is much smaller. Data from the 8,000-host network at LBL [20] indicates that the total number of active flows never exceeded 1200 in any one second. Results collected using Argus [21] from the Computer Science and Electrical Engineering network which connects more than 5500 *active* hosts are shown in Figure 6. We find that the maximum number of flows active in any one second over a period of 7 days during the end of January, a busy time, only crosses over 9000 once, and stays below 10000. The number of active flows seen in both the LBL network and the Stanford EE/CS network fit very easily into the NetFPGA OpenFlow switch's 32,000-entry exact match table.

Table 1: Summary of NetFPGA OpenFlow switch performance.

Pkt Size (bytes)	Forwarding (Mbps)	Full Loop (flows/s)
64	1000	61K
512	1000	41K
1024	1000	28K
1518	1000	19K

It is worth noting that at one point, the number of active flows recorded from a single IP address reached more than 50k active flows in one second for a period of two minutes. However, almost all the flows consisted of two packets, one that initiates the flow and one that ends it after 50 seconds. The host that generated all these flows was contacting an AFS server for backup. We suspect the first packet was a request that timed out, while the second packet was the timeout message. The large number of flows is due to the way Argus counts UDP flows. Even when the flow consists of only two packets spread out over 50 seconds, Argus still counts them as active flows. We do not include the results from this single host because this situation could have easily been remedied in an OpenFlow network using a single wildcard table entry. A more optimized version of the implementation can handle more than 64,000 entries, so even without the wildcard entry, we would still be able to run the network.

Forwarding: We ran two tests on our NetFPGA OpenFlow switch. First, to test the hardware's forwarding rate, we inserted entries into the hardware's flow table and ran streams across all four ports of the NetFPGA. This was done using a NetFPGA packet generator that can transmit predetermined packets at line-rate. A NetFPGA packet capture device audited the output from the OpenFlow switch to make sure we received the expected packets. The `forwarding` column in table 1 shows that our switch is capable of running at the full line-rate across 64, 512, 1024, and 1518 packet sizes.

New Flow Insertion: Second, we tested the rate at which new flows can be inserted into the hardware flow table. This answered the following question: Assuming an infinitely fast OpenFlow controller, at what rate can new flows be coming into the switch before they start getting dropped? The test was run by connecting the NetFPGA switch to an external host that continuously generated new flows. A simple controller implementing a static Ethernet switch [3] was run on the same machine as the switch so that the local OpenFlow switch manager and the OpenFlow controller could communicate through memory. We calculated the rate at which new flows were received by the NetFPGA and the rate at which new entries were created in the NetFPGA flow table. The results are summarized in the `full loop` column of table 1.

The main bottleneck in the system is due to the PCI bus—

[3] i.e. it uses static routes

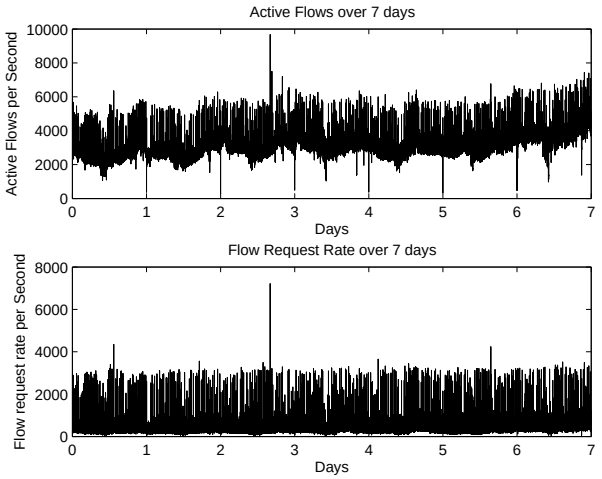

Figure 6: Recent measurements obtained from the Stanford EE and CS buildings. The top graph is the total number of active flows per second seen on the network over seven days, and the bottom graph is the number of new flows that are seen every second over the same seven days.

a single register read or write can take up to 1 μs. Since there is 1 register read and 10 register writes to insert a flow entry into the hardware, it takes at least 11 μs to insert an new flow. To make matters worse, the packet is also re-injected across the PCI bus via DMA into the hardware to be sent out. Another factor that reduces flow insertion rate is the high interrupt rate, especially with small packet sizes.

A more sophisticated hardware mechanism to mitigate the problem can create flow headers from the packets automatically and buffer them in the hardware so that if they need to be added into the flow table, all that is needed is a single register write. The hardware can also buffer packets and send summarized packets up via the software to the controller along with a pointer to the buffered packet. The OpenFlow protocol [18] specifies the mechanism with which a controller can ask a switch to inject a packet buffered locally on the switch back into the data path. Interrupt mitigation techniques can be used to reduce the interrupt rate.

The maximum new flow request rate seen in the Stanford EE/CS network shown in the bottom of figure 6 was less than 7000 flows/second, and this corresponds to the same maximum seen in active flows and was identified as being caused by an aggressive SSH troll. The rate at which new flow requests are generated from Stanford's network is well within the capabilities of the simple unoptimized NetFPGA OpenFlow switch implementation.

5.2 Complexity

To compare complexity, we look at the Output Port Lookup module in isolation for each of the OpenFlow Switch, the IPv4 router, the learning Ethernet switch, and the NIC. We divide up the module into three main parts: Logic, Primary Lookup, and Secondary Lookup. For all four designs, the Logic component contains no lookup logic or lookup tables. For the OpenFlow switch, the primary lookup is the 32,768-entry exact-match with its required hashing, while the secondary lookup is the 32-entry wildcard match along with the TCAM logic used. For the IPv4 router, the primary and secondary are the 1024-entry TCAM LPM with its associated lookup table and 32-entry CAM ARP cache lookups respectively. The Ethernet switch has only the 512-entry CAM MAC table lookup as a primary lookup, while the NIC has no lookups at all. The TCAM for the router was implemented using SRL16e while the CAM for the Ethernet switch as well as the CAM for the router's ARP cache were based on a dual-port BRAM implementation ([25], [24]). Since the 36 Mbit SRAM for the OpenFlow exact-match lookup is off-chip, it is not included in our results.

The build results for all four designs are summarized in Table 2 below. These results were obtained using Xilinx's implementation tools from ISE9.1i Service Pack 3. We give the results as LUT and DFF counts and Block RAMs used on the Virtex-IIpro, as well as a percentage of total LUT utilization.

The implementation results suggest that by way of logic utilization, an OpenFlow switch is between a basic IPv4 router and a basic Ethernet learning switch. A feature-rich IP router or Ethernet switch, however, is much more complex than the basic implementations we have. On the other hand, a basic OpenFlow switch when combined with the right OpenFlow controller can already handle many of a complex router or switch's additional features, such as access control.

From discussions with contacts at several switch and router vendors, we have concluded that a router with features similar to the Gates building router uses around 5M logic gates per port. This number is equivalent to our `Logic-Only` column, and includes the forwarding engine and switch fabric. If we compare this number to those from the NetFPGA OpenFlow switch implementation, the NetFPGA IPv4 Router implementation, and the NetFPGA learning switch implementation, we get the following results:

NetFPGA learning Ethernet switch : 141,885

NetFPGA IPv4 router : 515,802

NetFPGA OpenFlow switch : 167,668

Feature-rich router : 5,000,000

The logic utilization difference between a basic IPv4 router and a feature-rich router is almost 10x. On the other hand, the OpenFlow switch can provide many of the feature-rich router's functionality at 1/30 the logic gate cost.

It is difficult to compare the lookup technologies employed in the implementations of the designs used in Table 2. This is especially true since the results are based on a FPGA implementation where CAMs and TCAMs are not implemented in the usual 13 transistor/cell way. In addition, the OpenFlow switch's lookup table stores *active* flow information whereas a router or switch's lookup table stores information on all *potentially active* flows. To provide the same granularity of control OpenFlow provides, we expect that a router or switch's forwarding table size will have to increase exponentially with the number of hosts that connect through it. On the other hand, we expect the size of an OpenFlow switch's forwarding table size to increase a little faster than linearly.

As for software, we note that the current open-source implementation of OpenFlow uses 27378 lines of C code,

Table 2: Build results of the OpenFlow switch, the IPv4 router, the Learning Ethernet switch, and the NIC.

Function	OpenFlow Switch	IPv4 Router	Ethernet Switch	4-port NIC
Logic-only	1179 LUTs, 850 DFFs 2 BRAMs, 3% Area	2874 LUTs, 1245 DFFs 7 BRAMs, 6% Area	783 LUTs, 388 DFFs 2 BRAMs, <1% Area	38 LUTs, 8 DFFs 0 BRAMs, <1% Area
Primary	1881 LUTs, 1227 DFFs, 0 BRAMs, 5% Area	11125 LUTs, 2272 DFFs, 6 BRAMs, 60% Area	3267 LUTs, 748 DFFs, 98 BRAMs, 10% Area	N/A N/A
Secondary	7661 LUTs, 5501 DFFs, 16 BRAMs, 26% Area	623 LUTs, 391 DFFs, 7 BRAMs 2% Area	N/A N/A	N/A N/A

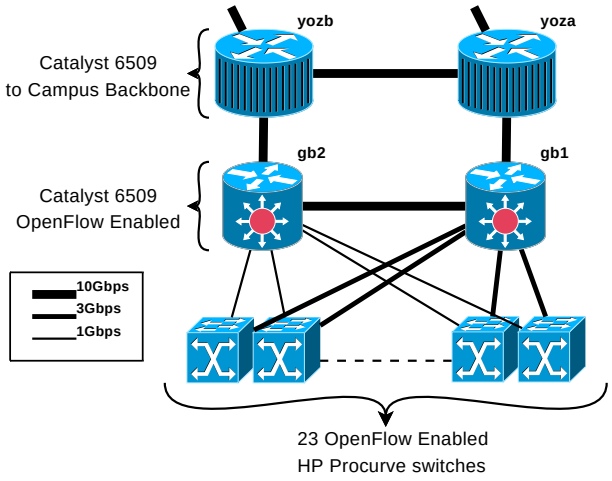

Figure 7: The OpenFlow deployment in the Gates building will consist of 23 HP ProCurve switches connected to two Cisco routers.

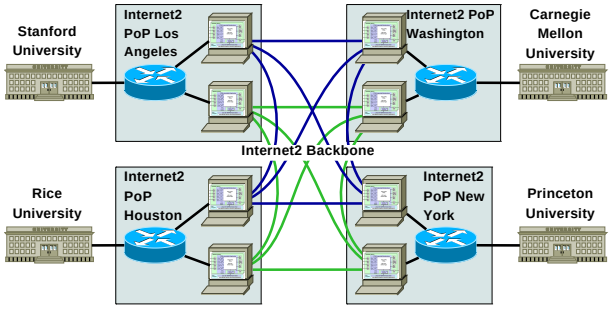

Figure 8: The NetFPGAs in Internet2 are connected in two parallel meshes. The Internet2 backbone deployment of NetFPGA will allow us to run OpenFlow across several university campuses.

whereas our implementation of a simplified OSPF (PW-OSPF [23]) uses 22395 lines of code both including comments. On the other hand, full-featured router software such as IOS [3], can reach millions of lines of code.

6. DEPLOYMENT

We will deploy both commercial OpenFlow switches and NetFPGAs that can run as OpenFlow switches in two buildings at Stanford, as well as into the Internet2 backbone. At Stanford, we will deploy modified HP ProCurve 5400 (model numbers 5406zl and 5412zl) Ethernet switches [7] and Cisco Catalyst 6500 series routers [2] to which experimental OpenFlow support was added to allow researchers to run experiments on the building's network. We have also deployed NetFPGAs in the Internet2 backbone allowing us to run OpenFlow over a large network with traffic from around the United States.

Stanford Deployment: Our Stanford Gates building OpenFlow deployment is shown in Figure 7. The Cisco routers connect 23 HP switches that in turn connect the Gates offices and machines. `gb1` is the main router while `gb2` is used in case of a failure. Work is underway to implement the management control for these OpenFlow switches. The Stanford deployment has been used in a demo at SIGCOMM 2008—A Demonstration of Virtual Machine Mobility in an OpenFlow Network.

Internet2 Deployment: The Internet2 NetFPGA deployment consists of 8 Dell PowerEdge 2950 2u servers with a NetFPGA installed. The machines are distributed in pairs to the following Internet2 PoPs: Los Angeles, Houston, New York, and Washington D.C. The NetFPGAs are connected in two parallel meshes—each NetFPGA connects to three other NetFPGAs at the other PoPs with dedicated circuits. The last NetFPGA port is connected to the Internet2 router in the PoP, allowing connection back to one of Stanford, Rice, Princeton, or Carnegie Mellon universities. In addition, a rack of 40 NetFPGA machines has been deployed at Stanford with direct connection to the Los Angeles PoP allowing further programmability and expansion of the network. Figure 8 gives a diagram of the deployment. At the time of writing, the NetFPGAs in Los Angeles and Houston were installed, while the New York and Washington D.C. installations were still underway.

7. RELATED WORK

Others have already proposed using centralized designs for security and management. As was mentioned before, OpenFlow is an abstraction of Ethane's datapath [1]. A prototype for an Ethane switch had been previously built on an older version of NetFPGA [11]. This work uses a more modular approach and implements OpenFlow, which is more general and flexible than Ethane's original datapath.

4D [5] advocates the division of the network architecture into four components, the decision plane, the dissemination plane, the discovery plane, and the data plane. Like OpenFlow, 4D advocates centralization of the decision making in the network. However, unlike OpenFlow, 4D does not provide fine-grained, per-flow control over the network. The OpenFlow protocol can be viewed as providing 4D's data plane, discovery plane, and dissemination plane.

Molinero-Fernández and McKeown [14] suggested TCP switching for implementing dynamic circuit switching in the Internet to eliminate switch buffers and make it easier to

build all-optical routers and provide better services. While TCP switching can be implemented with OpenFlow, OpenFlow can do more. It removes the decision-making from the datapath and delegates it to a central controller. Instead of using the initial TCP SYN packet to establish a circuit, the OpenFlow switch sends the SYN packet to the central controller. In addition to making routing decisions, the central controller can implement access control, accounting, decide on a service level, etc. All these decisions are made once for each flow in a central location as opposed to doing it hop-by-hop as in the TCP switch case.

ATM [22] follows a similar pattern where a virtual circuit is established across a network before packets traverse the network. While similar in that respect, OpenFlow is centralized, and does not necessitate per-packet modifications. OpenFlow uses the link, network, and transport layers to define a flow, while ATM operates on a single layer, namely the link layer. OpenFlow brings back the concept of a simple virtual circuit by using fine-grained flows. ATM switches need hard state since a circuit is established once at the beginning of a session, while OpenFlow switches can maintain soft state since they only cache information from the controller, and all packets of a flow are treated in the same way.

Lockwood *et al.* ([10], [9]) have designed the Field Programmable Port Extender (FPX) platform to develop reprogrammable and modular networking hardware. The FPX platform had more FPGA resources and memory to handle deep content-level processing of network traffic, whereas the NetFPGA is optimized to be a low-cost teaching and prototyping platform. The interface between NetFPGA modules are FIFO-based and are simpler than the Utopia-based ones used in FPX. The open-source gateware and software provided for NetFPGA allows for a gentler learning curve of hardware design for networking.

Click [8] implements a modular router datapath in software. NetFPGA aims to do for networking in hardware what Click did for networking in software — enabling innovation by lowering the barrier to implementing and extending networking components. Whereas Click uses both push and pull interfaces between modules, the NetFPGA reference pipeline uses only push. Click is not able to forward packets at full-line rate on commodity hardware, whereas a NetFPGA router can route at full-line rate across 4 1-Gbps ports even using outdated hardware.

8. CONCLUSION

OpenFlow provides a simple way to innovate in your network. It allows researchers to implement experimental protocols and run them over a production network alongside production traffic. We have implemented OpenFlow on the NetFPGA, a reprogrammable hardware platform for networking. Our implementation is capable of full line-rate switching and can easily accommodate all the active flows in a production network such as the Stanford University Gates Building.

The OpenFlow implementation on the NetFPGA exposed a few issues that designers should be aware of. The size of the flow header entry—currently 240 bits for the NetFPGA implementation—along with the actions for the entry can be a significant bottleneck. This problem is exacerbated when the packet is re-injected from the software into the hardware, using the same communication channel—the PCI bus in this case. A design that can buffer flow entries and packets in hardware, and only send summaries to the controller can help mitigate these effects. Despite these bottlenecks, the NetFPGA implementation can handle the traffic seen on the Stanford campus.

Some issues remain to be addressed as the protocol matures and the OpenFlow switch Type-1 is defined. These include how to protect the controller or the switch from denial of service attacks, how to manage an OpenFlow network so that multiple controllers can be running, how to extend OpenFlow to handle more than TCP/IP or UDP/IP traffic, and how to handle broadcasts elegantly. Other issues such as QoS and enabling dynamic circuit switching are still to be decided on.

While OpenFlow obviously suffers a few shortcomings, none of them are show-stoppers, and they allow the protocol to be simple enough that it has already been implemented in a few commercial switches and routers. We believe OpenFlow enables a new type of networking research as well as allowing researchers to run experimental protocols over real deployments.

9. ACKNOWLEDGMENTS

We wish to thank John Lockwood for leading the NetFPGA team, initial reviews of the paper, and helpful discussions and John Gerth for helping us get statistics on the Stanford network usage.

10. REFERENCES

[1] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, 2007.

[2] Cisco. Catalyst 6500 series overview website. http://www.cisco.com/en/US/products/hw/switches/ps708/.

[3] M. M. Coulibaly. *Cisco IOS Releases: The Complete Reference*. Alpel Publishing, 2000.

[4] W. H. Graves. All packets should not be created equal: the Internet2 project. Technical report, 1998.

[5] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, 2005.

[6] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, 2008.

[7] Hewlett-Packard. Hp procurve 5400zl series specification. www.hp.com/rnd/products/switches/ProCurve_Switch_3500yl-5400zl_Series/specs.htm.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[9] J. W. Lockwood. An open platform for development of network processing modules in reprogrammable hardware, 2001.

[10] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable network packet processing

on the field programmable port extender (FPX). In *FPGA*, pages 87–93, 2001.

[11] J. Luo, J. Pettit, M. Casado, J. Lockwood, and N. McKeown. Prototyping fast, simple, secure switches for Ethane. *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*, pages 73–82, Aug. 2007.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[13] G. Minshall, B. Hinden, E. Hoffman, F. C. Liaw, T. Lyon, and P. Newman. Flow labelled IP over ATM: design and rationale. *SIGCOMM Comput. Commun. Rev.*, 36(3):79–92, 2006.

[14] P. Molinero-Fernández and N. McKeown. TCP switching: exposing circuits to IP. *IEEE Micro*, 22(1):82–89, 2002.

[15] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: reusable router architecture for experimental research. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 1–7, New York, NY, USA, 2008. ACM.

[16] NetFPGA Team. NetFPGA website. http://netfpga.org/.

[17] Nicira. NOX repository website. http://noxrepo.org.

[18] OpenFLow Consortium. OpenFlow switch specification. Can be accessed at http://openflowswitch.org/documents.php.

[19] OpenFlow Consortium. OpenFlow website. http://openflowswitch.org/.

[20] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *IMC'05: Proceedings of the Internet Measurement Conference 2005 on Internet Measurement Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[21] QoSient. Argus network auditing website. qosient.com/argus.

[22] K.-Y. Siu and R. Jain. A brief overview of ATM: protocol layers, LAN emulation, and traffic management. *SIGCOMM Comput. Commun. Rev.*, 25(2):6–20, 1995.

[23] Stanford University. Pee-Wee OSPF Protocol Details. Can be found at yuba.stanford.edu/cs344_public/docs/pwospf_ref.txt.

[24] Xilinx. Designing flexible, fast CAMs with Virtex family FPGAs. http://www.xilinx.com/support/documentation/application_notes/xapp203.pdf.

[25] Xilinx. An overview of multiple CAM designs in Virtex family devices. http://www.xilinx.com/support/documentation/application_notes/xapp201.pdf.