UNIVERSITY OF AMSTERDAM

MSC SYSTEMS & NETWORK ENGINEERING

Performance Analysis of OpenFlow Hardware

By:
Michiel APPELMAN
michiel.appelman@os3.nl

Maikel DE BOER
maikel.deboer@os3.nl

Supervisor:
Ronald VAN DER POL
rvdp@sara.nl

February 12, 2012

Summary

OpenFlow is a protocol that defines an open control channel towards the data plane of multiple switches from different vendors. This allows for a scalable implementation when running a large network. There has been growing interest in OpenFlow within the networking industry, with the promise of new versions and features added by the *Open Networking Foundation* (ONF). However, there is a lack of research in the performance of the switches that support it and if their performance will actually allow for the promised scalability. This report aims to fill this void by benchmarking three different hardware implementations provided by SARA with respect to their performance and OpenFlow features. Tested platforms include a NetFPGA card, Pica8 OpenFlow on a Pronto switch and the Open vSwitch implementation on the same switch platform. Tests include looking into the lookup procedure by the switches, Quality of Service features and impact of failover flows. In the end it appears that although the most basic forwarding features are available in the implementations tested, scalability is a characteristic, which cannot yet be added to the OpenFlow feature set. While the performance scalability could be improved by optimizing the hardware design, one of the more significant bottlenecks seems to be the interoperability between OpenFlow firmware of switches, their hardware and the software used to control them.

Contents

List of Figures

List of Tables

1 Introduction

OpenFlow is a relatively new protocol designed and implemented at Stanford University in 2008. This new protocol aims to control the data plane of a switch, which has been physically separated from the control plane using a software controller on a server. The control plane communicates with the data plane through the OpenFlow protocol. This form of *Software Defined Networking* (SDN) allows researchers, administrators and operators to control their network with dedicated software and provides an *Application Programming Interface* (API) towards the forwarding tables of the switches from different vendors.

OpenFlow is based on installing flows in switches. A flow consists of a match and an action. A network packet travels into a switch and can be matched on several fields in the headers of various network protocols. The protocol describes several actions of which the most common one is to forward the frame. It can be forwarded out of all interfaces, encapsulated and sent to the controller, perform an action based on a flow in the switch's table, etc. Optional actions include enqueuing, which is used for *Quality Of Service* (QoS), dropping of the frame or modifying it to for example change the *Virtual* LAN (VLAN) ID on an incoming packet.[1]

Controllers are usually servers running the controller software which talks directly to the OpenFlow switch through the API. One of the more common Open Source controllers is 'NOX'. On top of this controller software developers can write their own modules, in case of NOX by using Ruby or C++.

The first versions of OpenFlow were developed at Stanford University, but since March 21 2011 the OpenFlow standard is being developed and maintained by the *Open Networking Foundation* (ONF).[1] Since the founding many large players in the networking industry have joined, amongst others Facebook, Google, Microsoft, HP, Intel, Cisco, Juniper and IBM.

1.1 Motivation

The OpenFlow protocol is relatively young and still under heavy development, but looks like a promising technology in the world of networking, SARA is actively involved in research of various OpenFlow related topics. One of the achievements of SARA's researchers is implementing IEEE 802.1ag Connection Fault Management as an OpenFlow controller component.[2] The author of this software is Ronald van der Pol, also supervisor of this project.

Although there has been growing interest in OpenFlow within the networking industry, there is a lack of research in the performance of the switches that support it, and if their performance will actually allow for the promised scalability. SARA asked us to fill this void by benchmarking their OpenFlow hardware to see how it would perform and scale.

1.2 Research

The conducted research is a result of four weeks dedicated work done as the first of two research projects in the masters course 'System and Network Engineering' at the University of Amsterdam.

1.2.1 Research Question

To summarize the goal of this research, the following research question will be answered:

> *"How can the scalability of OpenFlow switches be explained by their hardware design?"*

To answer the research question we will run a series of experiments on the hardware provided by SARA. In Chapter 2 the experiments are treated one by one, first giving an introduction to the experiment, then the

[1] https://www.opennetworking.org/membership/members

procedure followed by the results of the experiment. In Chapter 3 we will discuss the results, answer the research question and give some recommendations for people who want to use or research OpenFlow.

1.2.2 Scope

This research is aimed at measuring the performance and researching different functionalities of specific OpenFlow hardware. During the research we will not design any new applications to program switches or other OpenFlow related tools.

1.2.3 Related Work

In 2009[3] and 2010[4] several performance tests were done to measure amongst others the throughput and latency of OpenFlow in comparison to standard switching and routing. This past research focused more on the data plane performance while in this project different switches and the OpenFlow controller itself will be looked at in more detail.

2 Experiments

Three different OpenFlow platforms were tested during this project: *a*) a server with a NetFPGA card running an OpenFlow 1.0.0 implementation; and two Pronto 3290 switches capable of running *b*) an OpenFlow 1.0.0 implementation by Pica8 ported from the Indigo project[5]; and *c*) the Open vSwitch software version 1.2.2, which has its own typical implementation of OpenFlow. The NetFPGA PCI-card is a 4-port Gigabit card aimed at research and education uses.[6] It is build as an open platform with an *Field Programmable Gate Array* (FPGA) that can be programmed to specific needs. The Pronto switch is also build as an open switching platform.[7] The 3290 model is an *Top of Rack* (ToR) switch with 48 Gigabit ports and 4 10GbE ports. It supports multiple firmware versions with different features, including a standard Layer 2 switching firmware and the two aforementioned OpenFlow versions. The Open vSwitch implementation is typical because it has implemented some OpenFlow 1.1.0 features, and even some 1.2 features already.

The specifications of these units, their OpenFlow implementations and performance will be discussed in depth in this section using a series of tests.

2.1 Lookup Procedure

OpenFlow switches use the flows they have received from their controller to make forwarding decisions. Ingress frames can be matched exactly to a flow, matched to a flow with wildcard fields or they will not match to any flows at all.

Flows installed in an OpenFlow switch are stored in flow tables. These tables contain the information on which a frame can match and attached is an action to be performed by the switch. In total the OpenFlow 1.0.0 specification includes 12 fields that can be matched upon. A flow can be created for a specific stream of traffic, this means that all the fields will match the frames to that flow, this means that the input port, source and destination MAC address, IP address, TCP/UDP port, etc. must all match with that flow. These flows are stored in a '`hash`' table because the 12-tuple is hashed and then stored as an index in a table for fast lookups. If one or more of the fields are wildcarded, i.e. can match with any value, the flow is usually stored in a '`linear`' table, to be looked at after the '`hash`' table in a linear fashion.

The hashes of the exact flows are typically stored in *Static* RAM (SRAM) on the switch. This memory allows for an indexed table of the hashes and a fast lookup procedure by the switch. One drawback of this type of memory is that it is usually off-chip causing some latency.

Often located on the switching chip is another type of memory that is used to store the `linear` table, called *Ternary Content-Addressable Memory* (TCAM). It contains a tertiary state to match upon next to the binary `0` and `1`, namely the `X` state, which functions as a mask.[8] TCAM is most efficient for the fields that are wildcarded in a certain flow. TCAM is usually located on-chip, but because it's relatively expensive it's not possible to implement large amounts of it on the switch.[9]

Using some basic techniques this test will try to see how these functionalities are actually implemented in the hard and software. Key questions here are:

1. How many exact/wildcard flows can the switch support?
2. What happens when the tables are full?
3. What is the impact on throughput on smaller frame sizes using linear lookups?

2.1.1 Procedure

To measure how much flows could be installed in the switch a script was written, as listed in Listing 5 in Appendix A. The flow tables in the switch were monitored as they filled up to see how many flows they could store. For the wildcard flows a different flow was used that only included the Ethernet *Destination Address* (DA) and *Source Address* (SA), and incremented the EtherType value in the match statement.

An Anritsu MD1230A[2] traffic tester was used to run an RFC2544 test[10] to measure the throughput and latency of a wildcard flow, that is from one SA to the another port on the device with another DA. This test increments the frame size with each step to see how the switch handles it, for Ethernet the frame sizes are given in paragraph 9.1 of the RFC. As the test only covers Layer 2 and 3 of the OSI model, it is impossible to create an exact match, which always includes a Transport protocol source and destination port.

2.1.2 Results

It should be mentioned that the Open vSwitch implementation only uses one table called a '`classifier`'-table and classifies flows in its own way to see what fields it should wildcard.[11] This essentially means that all flows are given some sort of hash in memory, with the software deciding on the wildcarded fields. The Open vSwitch team also states that it is not possible anymore to make complete exact matches for OpenFlow 1.0.0 anyway because Open vSwitch supports fields that the OpenFlow specification does not.[11]

The NetFPGA card on the other hand, includes an extra table, which actually is a copy of the hardware tables.[12] Using exact match flows this '`nf2`' table is filled up first and after that the '`hash`' table is used as well. Of the wildcard flows the first 24 are put in the '`nf2`' table and after that the '`linear`' table is filled.

	NetFPGA	OpenFlow	Open vSwitch
Displayed Maximum Hash Flows	32792 + 131072	131072	*N/A*
Actual Maximum Hash Flows	31421 + 131072	1664	*N/A*
Displayed Maximum Linear Flows	100	100	1000000
Actual Maximum Linear Flows	124	100	1664

Table 1: Available space in OpenFlow tables in different implementations

The aforementioned script was run on the three different platforms and Table 1 represents the data gathered. The displayed maximum is returned by the `dump-tables` command to see the contents of the tables. As discussed the NetFPGA stack has two limits for its hash flows, the first one for the '`nf2`' table and the second for the exact '`hash`' table to match against. The displayed maximum of 32,792 flows isn't reached in the '`nf2`' table though. The displayed maximum for linear flows is only 100 but an extra 24 are installed in the '`nf2`' table. This can be seen by installing flows one-by-one and dump the tables of the flows after each install.

The standard OpenFlow implementation on the Pronto switches displays its maximum number of flows at 131,072 but the maximum appears to be only 1,664. This value becomes even lower when wildcard flows are installed: with 1 wildcard flow installed the maximum is 1,526 flows, 128 flows lower. Between 2 and 100 wildcard flows the maximum becomes another 128 flows lower at 1,408. The differences between the displayed and actual values are caused by limitations in the hardware design that the software isn't aware of. The exact hardware properties of the Broadcom chip in the Pronto switches are not disclosed and the OpenFlow datapath implementation on this platform is also a closed-source effort.[13]

Maximum hash flows	
No wildcard flows	1,664
One wildcard flow	1,536 *(-128)*
2 – 100 wildcard flows	1,408 *(-256)*

Table 2: Available space in OpenFlow hash tables with wildcards on Pica8 OpenFlow

Open vSwitch shows a similar value but displays an initial value of 9 active flows in its '`classifier`' table, although no flows are displayed when the switch is polled. This probably is an artefact of the software style implementation of Open vSwitch. Filling up the table finally yields 1,673 as the maximum number of flows, the same as the standard OpenFlow implementation (subtracting the 9 default flows).

When the tables in the switch are full, the OpenFlow 1.0.0 states in section 4.6 that the switch should send an error message to the controller with this reason.[1] For frames coming from unknown flows, this would

[2] http://www.anritsu.com/en-US/Products-Solutions/Products/MD1230A.aspx

mean that they will be dropped. Unless the controller processes this error and instead of a flow modification message sends a message towards the switch indicating it should directly output the frame to a certain port (a so called `packet_out` event). This is not scalable though, as it would mean that every frame would have to be processed by the controller.

The message that a switch sends to the controller to inform it about an unknown flow is called a `packet_in` event message. This message includes the characteristics of the flow and a by the controller predetermined amount of payload. Using a tool called OFlops the latency between a frame sent and a `packet_in` event received from a switch can be determined. Figure 1 shows the setup needed to do so, a host with two Ethernet interfaces, one connected to a switch port and one to the management port. The interface connected to the switch port starts to send unknown flows and listens on the control channel port for incoming `packet_in` messages.

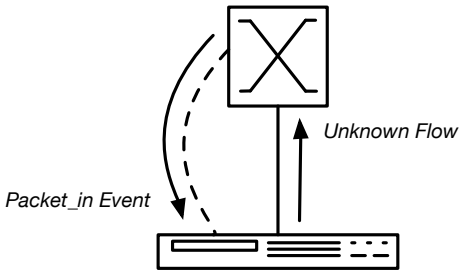

Figure 1: OFlops packet_in latency measurements

Testing this was not possible with the Open vSwitch implementation because it uses a version of the OpenFlow stack that is not compatible with the OFlops software, and thus the software will not recognize the incoming messages. After running the test for four times for 120 seconds on the NetFPGA and Pronto OpenFlow implementation, they showed an average latency of respectively 40.1 and 40.5 milliseconds.

The RFC2544 was used to test maximum supported throughput and latency with different frame sizes. For each implementation three trials were run for each frame size for 60 seconds. The Pronto switches could forward without packet loss at 100% of line rate at 1Gb/s with both the standard OpenFlow and Open vSwitch implementation with all Ethernet frame sizes. After a binary search between 100% and 90% by the Anritsu with a resolution of 0.1% the NetFPGA card reached 99.9% throughput on all frame sizes.

Figure 2 shows the results of the RFC2544 latency tests. The NetFPGA card displays a relatively low latency here, with a linear increasing pattern. The two Pronto implementations show a different graph, suddenly rising at 256 byte frames and then somewhat decreasing. The standard OpenFlow implementation on the Pronto also shows to be approximately 0.150 microseconds faster than the Open vSwitch firmware. A possible explanation might be the software-based '`classifier`' table and the overall larger stack that comes with Open vSwitch.[14]

2.2 Installing Flows

As networks grow they will need to handle more an more flow changes. The scalability here depends on how many flows a switch can handle and how fast flows can be installed.

In a regular OpenFlow switch one will find two or three flow-tables depending on hard- and software. When querying a switch for the amount of flows it can handle one will receive results including the maximum amount of flows and the amount of active flows. The amount of flows each tested device can handle is covered in section 2.1.

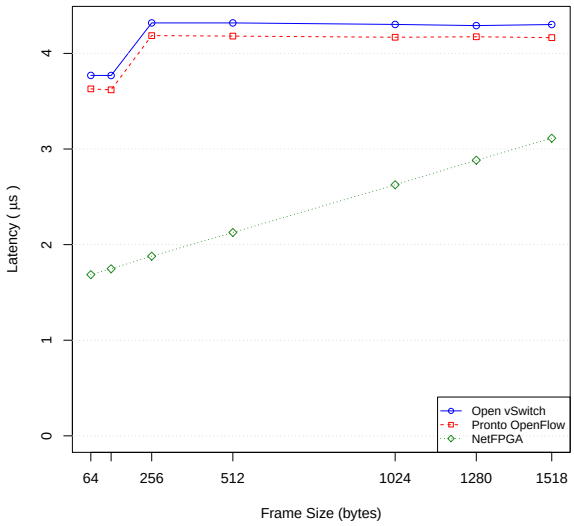

Figure 2: Average latency using wildcard flows

2.2.1 Test Set-up

To measure the speed we use the OFTest framework [3] to act as an OpenFlow controller. We used a modified version of the basic capabilities and capacities test to time the installation of flows. The modified part of the source can be found in listing 6. When a switch connects to the controller the standard negotiation is done and then flows are installed as fast as possible. By letting the switch connect to the controller and wait until the negotiation is done we get more accurate results.

2.2.2 Procedure

To see how fast flows are installed we pointed the switch to the server running OFTest, acting as the controller. When the switch connects to a controller it first sends a `Hello` packet, then the switch sends a `Hello` packet back, then the controller sends a `Features Request` packet, the switch should reply with a `Features Reply`. Then the switch sends a `Vendor` message, if the `Vendor` message is not understood by the controller as in our case the controller sends a `Error` packet back. Also the controller sends a `Flow Mod` packet to clear all flows from the switch. This complete process can be found in Figure 3.

After the switch and controller are done with the initial setup we start a timer and the controller starts sending flows as fast as possible to the switch. When the flow table of a switch is full the switch replies with a `Flow Mod Error` packet we stop the timer and count the amount of flows installed. We ran this test 10 times for each type of hardware/software configuration.

2.2.3 Results

In Table 3 one can see the time it took to install the maximum amount of flows for a specific table, the NetFPGA hash flow table is not completely filled due to limitations in the Oftest test. For this experiment it was also not possible to do tests with the Open vSwitch implementation because of an incompatible OpenFlow version in Open vSwitch.

In Table 4 the average time to install a single flow is shown. It is clearly visible that the NetFPGA is much faster installing a single linear flow compared to a single hash flow. The data regarding the hash flow on Net-

[3] http://www.openflow.org/wk/index.php/OFTestTutorial

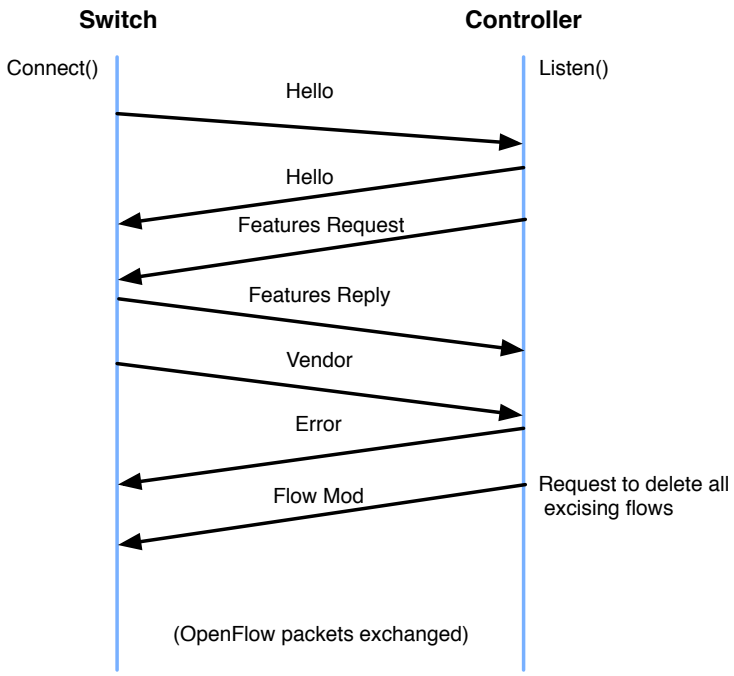

Figure 3: Packet flow between switch and controller when switch connects

	Min	Max	Average
Install 100 linear flows Pica8 OpenFlow	0.13	1.18	0.43
Install 1644 hash flows Pica8 OpenFlow	3.17	3.88	3.34
Install 100 linear flows NetFPGA	0.08	0.083	0.079
Install 65542 hash flows NetFPGA	401.31	402.66	401.75

Table 3: Time in seconds to completly fill a table with flows from a controller in a OpenFlow switch

FPGA could be improved because the timing was done over a huge set of flows, which adds extra overhead. We suggest to rerun this test with an even amount of flows for every piece of hardware.

For the Pronto switch running the Pica8 version of OpenFlow one can see that installing a single linear flow takes almost double the time of a hash flow.

	Pica8 OpenFlow	NetFPGA
Install 1 linear flow	0.004329	0.000785
Install 1 hash flows	0.002029	0.006130

Table 4: Avarage time in seconds to install a single flow from a controller in an OpenFlow switch

2.3 QoS Features

In previous (pre-1.0.0) versions of Open Flow QoS was not supported. Since OpenFlow 1.0.0 there is one QoS feature (minimum rate) described in the specification [1]. In other OpenFlow documentation QoS is also known as 'slicing' [4]. Key questions about QoS and OpenFlow for this research are:

[4] http://www.openflow.org/wk/index.php/Slicing

1. Does the hardware provided by SARA support the new QoS feature?
2. Does the QoS minimum rate function works as expected?

2.3.1 Test Set-up

IPerf[5] was used to measure maximum *Transmission Control Protocol* (TCP) and *User Datagram Protocol* (UDP) bandwidth performance. It allows the tuning of various parameters and UDP characteristics and reports bandwidth, delay jitter, datagram loss. One server runs with a Gigabit Ethernet card and two IPerf server processes on two different UDP ports. Two laptops with gigabit Ethernet cards will connect both to a different server process through the OpenFlow switch as shown in Figure 4. Both laptops will send UDP streams send to the IPerf server at 1000 Mbit/s. The QoS rules are applied on the port facing the server.

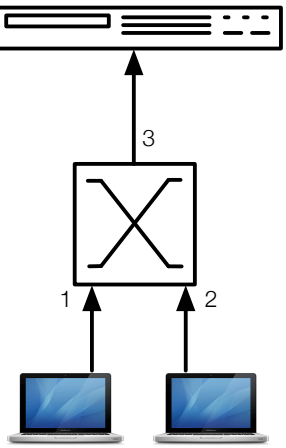

Figure 4: *Quality Of Service* setup

2.3.2 Procedure

First we will create two QoS queues using the `dpctl mod-queue` command giving the first queue 70% and the second 30% of the bandwidth as minimum rate. Second we install some basic flows to make sure *Address Resolution Protocol* (ARP) broadcasts and replies can get to the correct destination. Last thing we do is create flows for *Internet Protocol* (IP) address to port mappings, this is also the point the QoS queues are assigned to a specific flow.

The command list below shows how the flows and queues are setup on the NetFPGA OpenFlow implementation:

```
dpctl mod-queue unix:/var/run/test 3 1 300
dpctl mod-queue unix:/var/run/test 3 2 700

dpctl add-flow unix:/var/run/test arp,in_port=1,idle_timeout=0,actions=output:3
dpctl add-flow unix:/var/run/test arp,in_port=2,idle_timeout=0,actions=output:3
dpctl add-flow unix:/var/run/test arp,in_port=3,idle_timeout=0,actions=output:1,output:2
dpctl add-flow unix:/var/run/test icmp,in_port=1,nw_dst=192.168.0.3,idle_timeout=0,actions=output:3
dpctl add-flow unix:/var/run/test icmp,in_port=2,nw_dst=192.168.0.3,idle_timeout=0,actions=output:3
dpctl add-flow unix:/var/run/test icmp,in_port=3,nw_dst=192.168.0.1,idle_timeout=0,actions=output:1
dpctl add-flow unix:/var/run/test icmp,in_port=3,nw_dst=192.168.0.2,idle_timeout=0,actions=output:2
dpctl add-flow unix:/var/run/test in_port=1,ip,nw_dst=192.168.0.3,idle_timeout=0, \
            actions=output:3,enqueue:3:1
dpctl add-flow unix:/var/run/test in_port=2,ip,nw_dst=192.168.0.3,idle_timeout=0, \
```

[5] http://sourceforge.net/projects/iperf/

```
            actions=output:3,enqueue:3:2
dpctl add-flow unix:/var/run/test ip,in_port=3,nw_dst=192.168.0.1,idle_timeout=0,actions=output:1
dpctl add-flow unix:/var/run/test ip,in_port=3,nw_dst=192.168.0.2,idle_timeout=0,actions=output:2
```

2.3.3 Results

The first experiment will be executed on the NetFPGA card. We add two queues, first one with minimum rate 30% and one with a minimum rate of 70% as described above. We run IPerf from both clients sending around 750 Mbit/s of UDP data per client to the server. Because the link between the switch and the server is max 1Gb the QoS rules should provide a minimum rate speed for both flows. The results are shown in Figure 5. As we can see that the speed for both UDP streams does not come above the 105 Mbit/s.

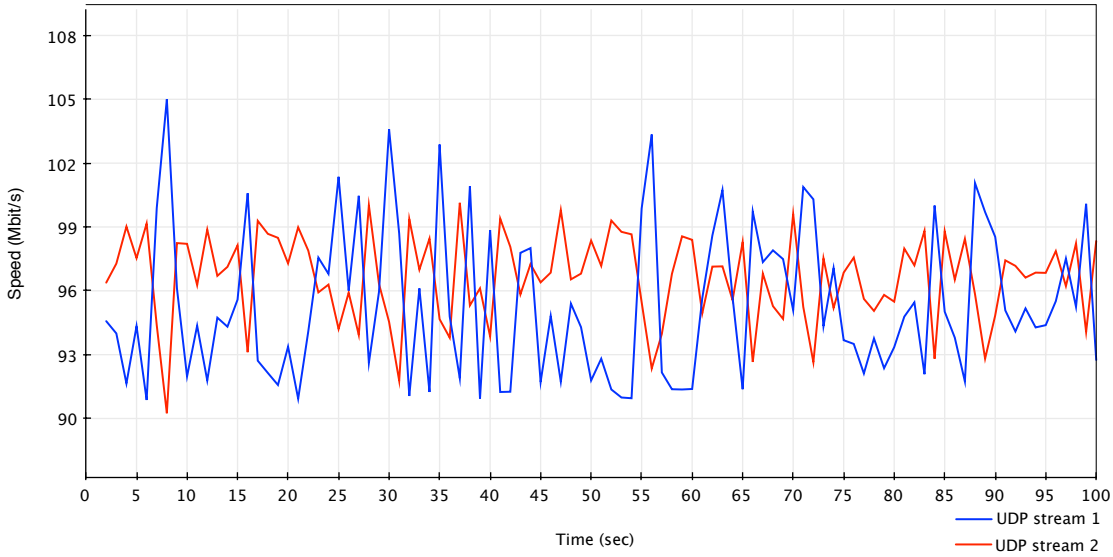

Figure 5: Two *Quality Of Service* queues enabled

The second experiment is also executed on the NetFPGA card. We add a single queue with a minimum rate of 70%. Again we send two UDP streams at 1000 Mbit/s, the results are shown in Figure 6. Stream 2 has queue for minimum 70% assigned, effectively the speed for stream 2 does not reach speeds over 100 Mbit/st.

When a queue is set for a specific stream that stream cannot reach higher speeds than 100 Mbit/s. According to the OpenFlow NetFPGA wiki page slicing is not supported: "Slicing feature is not supported" [6], supporting the results that were collected.

Second the Pronto switch with the Open vSwitch firmware was tested. Installing the flows as described above, we receive errors because commands do not exist. According to a blog post by Pronto "QoS and queue for port config, and flow match of tunnel id are not implemented." [6]

The Open vSwitch implementation has built-in QoS features. [7] When using the Open vSwitch features one obviously does not have the benefits of OpenFlow, it is not possible to let the controller add, remove and resize minimum rates automatically. Testing the Open vSwitch QoS features is out of scope for this research.

Finally the Pronto switch with the Pica8 firmware was tested. The `dpctl mod-queue` command exists but when we try to bind the queues to flows they don't show up when running the `dump-queues` command. When we check product page of the Pronto switches at Stanford University it says that slicing/enqueue is supported by Stanford Reference Software [8]. In the Pica8 firmware they used a port of the Indigo project.[9] In

[6] http://prontosystems.wordpress.com/2011/06/09/announce-ovs-support/
[7] http://openvswitch.org/support/config-cookbooks/qos-rate-limiting/
[8] http://www.openflow.org/wk/index.php/OpenFlowHWRoadmaps
[9] http://www.openflow.org/wk/index.php/IndigoReleaseNotes

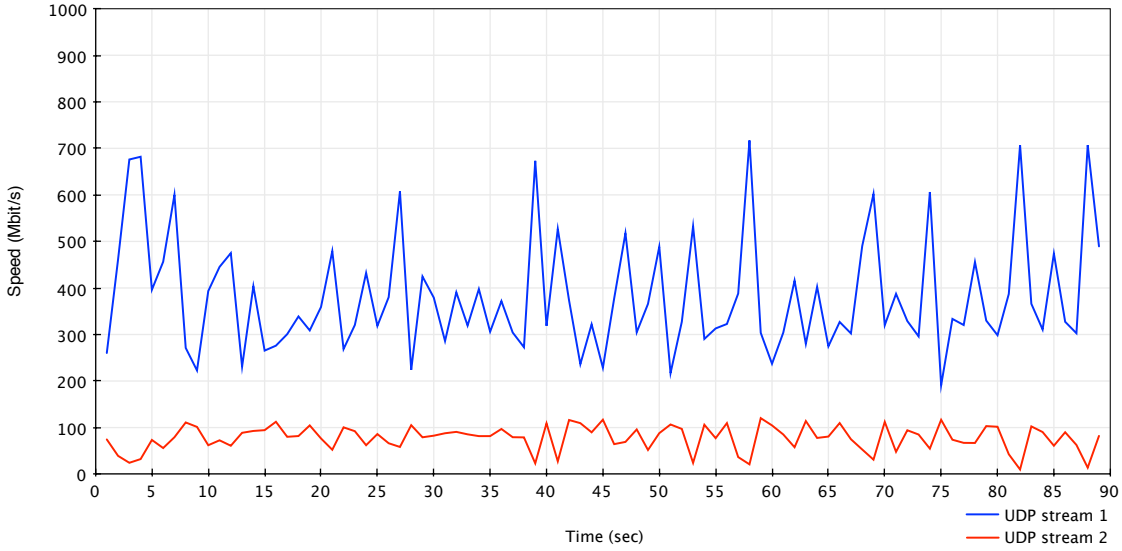

Figure 6: One *Quality Of Service* queue enabled

the Indigo release notes we find the following "Queues may be created using `dpctl` and the enqueue action is supported, though not completely verified."

After testing three different OpenFlow implementation on there QoS capabilities we must conclude that the standard describes QoS but implementations are very poor. NetFPGA does not work as expected (QoS is not supported), Open vSwitch does not implement OpenFlow QoS and on the Pronto switch running Pica8 firmware QoS is also not yet implemented.

2.4 Port Mirroring

A common feature in layer 2 Ethernet switches is to 'mirror' a port. This means that the egress frames destined for a port, are also sent out another port (or other ports). With OpenFlow matching can even be done on a specific TCP-stream and thus forward the frame to one or more additional ports. Depending on the design hardware this could have an implication on the throughput the switch can handle. When the ports share a common switch fabric, mirroring will use an extra portion of the bandwidth of the bus, limiting the other at sub-linerate.

2.4.1 Test Set-up

To test if this is the case for a certain platform, an OpenFlow switch can be connected port-to-port to a default Layer 2-switch with VLANs on every port as is shown in Figure 7. This is done to let the interfaces appear as 'up' to the OpenFlow switch so it will actually send traffic there and not blackhole it. For the Pronto this means 47-ports are connected to another Layer 2-switch, leaving one port for input. Also connected to the OpenFlow switch is a host that generates IPerf test traffic and a server to receive and log that traffic.

2.4.2 Procedure

A 1000 Mbit UDP stream is sent to the server for 180 seconds, which measures the actual bandwidth, the jitter of the frames and how many frames were lost. This test is first run over a direct connection between two hosts to set a baseline for the other tests. After that over one of the three platforms with a one-to-one flow and finally mirrored to all other ports. The 'mirror' flows are installed using the `dpctl` or `ovs-ofctl` containing one input port and all the other ports as output ports. Should mirroring of the port to 47 others

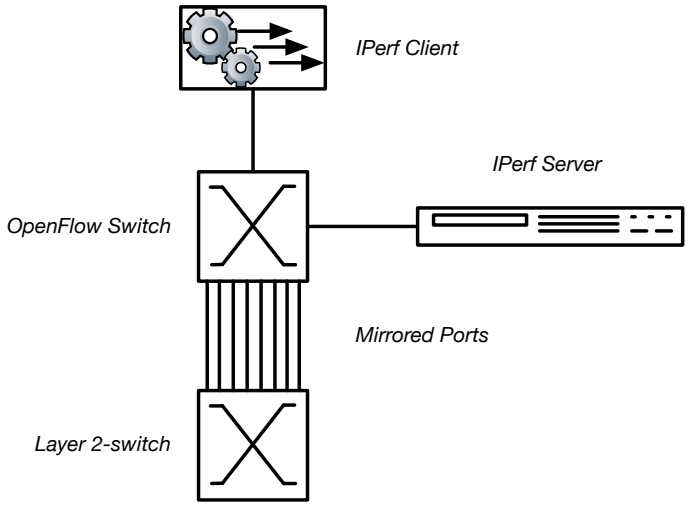

Figure 7: Test setup to measure port mirroring implications using IPerf

have a considerable impact on the traffic, a binary search may be used to see at what mirror-port count the impact occurs.

2.4.3 Results

The measurement between two hosts reported an average throughput of 695.75 Mbit/sec by IPerf. This would indicate that full line rate might not be possible but this might be caused by limits in the networking, *Input/Output* (IO) or *Operating System* (OS) stack of the hosts (Mac OS X 10.7) as IPerf is actually a memory-to-memory performance test. The result was established using a 512 Kbyte UDP buffer size, but even with an 8 Mbyte buffer full line rate was not achieved. Also, in Section 2.1.2 it was already concluded using the RFC2544 test that the Pronto switch will run at line rate. Thus this result and the same test settings will be used as a baseline to compare the performance of the OpenFlow hardware.

Comparing the other results from Table 5 to this baseline, it can be seen that there is some effect on the effective throughput. Firstly the throughput decreases with a switch in between, having a more negative effect on the Open vSwitch implementation. When the flow is later mirrored though, Open vSwitch performs better than without mirroring, while the NetFPGA and Pica8 OpenFlow implementations lose a little more throughput.

	NetFPGA	Pica8 OpenFlow	Open vSwitch	Direct Connection
Without Mirroring	687.6 Mb/s	685 Mb/s	668.7 Mb/s	695.75 Mb/s
Mirroring to 47 ports	666.2 Mb/s	650.6 Mb/s	682.4 Mb/s	*N/A*

Table 5: Throughput as reported by IPerf

Because of the limited tests carried using IPerf this result is not conclusive but it has shown that mirroring ports using OpenFlow flows has a measurable effect on the throughput of the generated test traffic. The test might be run again at a later time using an Anritsu traffic generator for more precise measurements.

2.5 Failover Speed

In theory and presentations given about the new protocol, OpenFlow promises to quickly adapt to changes in a dynamic network environment. This learning and switching to a new path could even happen in the middle of a running stream towards such a host. But what are the implications of such a swap? Is there any frame loss? Does it affect latency?

2.5.1 Test Set-up

To test such a situation at least two OpenFlow switches and a host are needed. In this case an Anritsu traffic tester was used. In Figure 8 a schematic representation of the setup is given, it features two links between the switches and a single link from each switch towards the Anritsu.

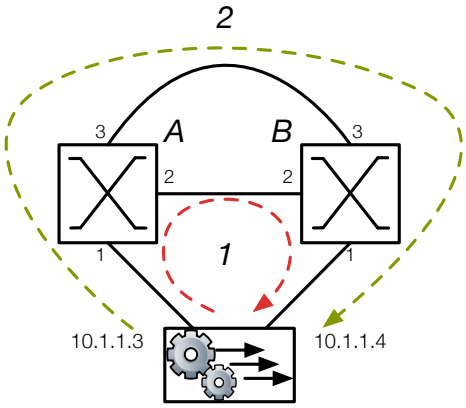

Figure 8: Setup for failover measurements. The swap will occur on switch *A* from path *1* to path *2*

2.5.2 Procedure

At first flows are added to allow the first interface of the Anritsu to reach the second interface over the switches. ARP packets also need to flow the other way around so flows are also added that way.

The flows installed on switch *A* are as follows:

```
ovs-ofctl add-flow br0 arp,in_port=1,idle_timeout=0,actions=output:2,output:3
ovs-ofctl add-flow br0 arp,in_port=2,idle_timeout=0,actions=output:1
ovs-ofctl add-flow br0 arp,in_port=3,idle_timeout=0,actions=output:1
ovs-ofctl add-flow br0 icmp,in_port=1,nw_dst=10.1.1.4,idle_timeout=0,actions=output:2
ovs-ofctl add-flow br0 icmp,in_port=2,nw_dst=10.1.1.3,idle_timeout=0,actions=output:1
ovs-ofctl add-flow br0 icmp,in_port=3,nw_dst=10.1.1.3,idle_timeout=0,actions=output:1
ovs-ofctl add-flow br0 ip,in_port=1,nw_dst=10.1.1.4,idle_timeout=0,actions=output:2
ovs-ofctl add-flow br0 ip,in_port=2,nw_dst=10.1.1.3,idle_timeout=0,actions=output:1
ovs-ofctl add-flow br0 ip,in_port=3,nw_dst=10.1.1.3,idle_timeout=0,actions=output:1
```

The flows installed on the other switch are similar in statements but point the other way.

After the flows are installed, connectivity is checked using ICMP messages. When the connection has been verified a stream is setup on the Anritsu running at the line rate that is known to have no frame loss (see Section 2.1.2). The stream consists of 100.000.000 test frames containing a sequence number that the Anritsu uses to check the sequence and latency.

This stream is started and after 30 seconds the flow is modified on switch *A*, moving the path towards path 2 (ports 3 on the switches in Figure 8).

2.5.3 Results

When running this test on the Pica8 OpenFlow firmware the Anritsu detected a single sequence error and an average of 121.5 lost frames. A sequence error is given whenever a non-consecutive frame is received after another frame. In this case it would mean that 120 frames were lost between two frames.

The Open vSwitch on the other hand registered an average of 7,952 sequence errors, meaning the frames were received in a significantly different order. It also showed frame loss, which varied greatly, in some

trials the Anritsu even registered a negative frame loss, i.e. more frames were received than there were sent. An explanation for this behaviour may be that the switch, while swapping the path, mirrored the ports for a short period of time and frames were being duplicated. To test this theory, instead of just modify the flow to move to another port, the flow was deleted and immediately added. Unfortunately this led to even more frame loss (4,593), but a lower count of sequence errors (4,465).

The test with the NetFPGA hardware gave more encouraging results, dropping not a single frame and delivering them all in order. This might be explained by the programming of the NetFPGA card which specifically aimed at OpenFlow.

The latency was measured once every second, an interval too large to actually measure the hit on the fast swapping of the path. Thus there were no significant differences observed in these values during these tests. To accurately measure these values two ITU-T Y.1731[15] test devices exchanging Delay Measurement Messages every 100 milliseconds might be used.

	NetFPGA	Pica8 OpenFlow	Open vSwitch
Frame Loss	0	121.5	-1740 – 1898
Sequence Errors	0	1	7,952

Table 6: Results for failover test on different platforms.

2.6 Performance Overhead

OpenFlow decouples the control plane logic (software) from the data plane (hardware) of a switch. By decoupling the forwarding logic from the data-plane one can achieve great flexibility in their networks and a lot of advantages come to mind when decoupling the hardware and software, but what is the impact on performance? Does OpenFlow add a big amount of latency when switching packets or is the drawback negligible?

2.6.1 Test Set-up

When a switch 'knows' a flow and has it stored in his SRAM or TCAM there is no switching latency except the latency the hardware adds, this latency is almost always specified by the manufacturer [7]. What happens if the switch does not yet have the flow for a specific device in his SRAM or TCAM? A normal non OpenFlow switch will buffer the incoming packet and flood a broadcast message out of all ports except the ingress port to learn about all connected devices. When the switch receives the mac addresses of all connected devices it stores them in SRAM and then sends out the buffered packet.

An OpenFlow enabled switch also needs to buffer the packet, but needs to access the external controller to ask what to do with the unknown frame. In this experiment we assume the OpenFlow switch will behave as a normal L2 learning switch by using the NOX `pyswitch` component. The pyswitch component makes all switches connected to the NOX controller standard L2 learning switches without any spanning tree like functions. [10]

First we will test the Pronto with the Xorplus (non-OpenFlow) firmware as a baseline, then we try to run the same experiment with the Pica8 firmware enabled. This experiment is also not possible to perform with the Open vSwitch firmware because of the incompatible OpenFlow version in Open vSwitch.

2.6.2 Procedure

We connected the Anritsu with two ports in a switch so we can send special test packets from the third port of the Anritsu to the fourth port. The propagation delay caused by the length of the network cables is negligible. For the tests with Xorplus firmware we connected the switch with a console cable to a laptop

[10] http://noxrepo.org/pipermail/nox-dev/2011-November/008248.html

to check the ethernet-switching table. See Figure 9 for the tests with OpenFlow switches we connected the switch to a NOX controller Figure 10.

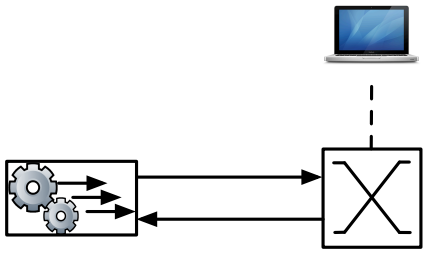

Figure 9: Xorplus firmware switch connected to laptop with serial cable for diagnostics

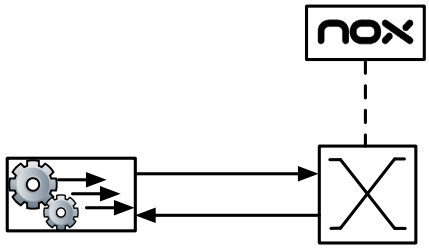

Figure 10: OpenFlow Pica8 firmware. Switch connected to a NOX controller

2.6.3 Results

Before we start testing the OpenFlow enabled devices we begin with a baseline on a Pronto switch running the non OpenFlow Xorplus firmware. We make sure the ethernet switching table is completely empty by running `run clear ethernet-switching table all`.

We started sending timestamped test packets (60-bytes) at full line rate from the third port of the Anritsu to the fourth port and vice versa . We did the test three times, the results can be found in Table 7.

First try	Port 4 to port 3	Port 3 to port 4
Max	7.1 µs	7.7 µs
Min	6.4 µs	7.1 µs
Average	7.0 µs	7.6 µs
Second try		
Max	7.1 µs	7.7 µs
Min	6.7 µs	7.4 µs
Average	7.0 µs	7.5 µs
Third try		
Max	7.1 µs	7.7 µs
Min	6.1 µs	6.9 µs
Average	7.0 µs	7.5 µs

Table 7: Xorplus latency in microseconds

We assume one will experience the most latency when the source mac address is not in the TCAM or SRAM so we consider this the time before the switch learned the destination mac address. The average max latency is

from port 4 to 3 is 7.1 µs from port 3 to 4 this is 7.6 µs . When the source mac is learned the average latency from port 4 to 3 is 6.3 µs and from 3 to 4 is 7.1 µs. The Pronto website for the 3290 switch says: "Switch latency: less than 1us for 64-byte frames when non-blocking." According to our tests the less then 1 µs is not achieved. We cannot explain the difference in latency between sending a packet from 3 to 4 and from 4 to 3.

	Port 4 to port 3	Port 3 to port 4
Max	7.1 µs	7.7 µs
Min	6.4 µs	7.1 µs

Table 8: Xorplus avarage latency in microseconds

After we complete the benchmark on the Xorplus firmware we reboot the Pronto switch to use the Pica8 firmware, also we startup the NOX controller in verbose mode with the pyswitch component. We configure the switch to automatically connect with the NOX controller at a specific IP address. When we see the switch is connected we start sending the test packets from the Anritsu from one port to the other. The test packets send from the one port were not detected as test packets anymore so we could not measure the latency.

We build the setup as found in Figure 11. We send a test packet out of port 3 from the Anritsu (A) to port 1 on a hub (B). The packet gets mirrored and send out of port 2 to the OpenFlow enabled switch (D) and out of port 3 to a laptop (C) running Wireshark capturing the packets. We also ran Wireshark on the Anritsu capturing all incoming packets.

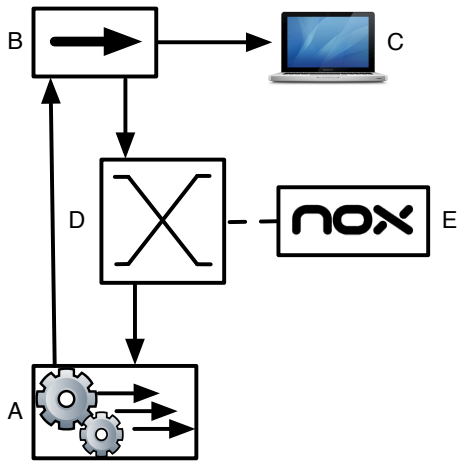

Figure 11: Performance overhead debug setup

Before the packet is send to the OpenFlow switch it is captured at the laptop it looks as in Listing 1, a 60 bytes on the wire ethernet frame. This is a frame as we expect it to be once it comes back in again at the Anritsu on port 4.

```
1 Frame 2: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
2 Ethernet II, Src: Beckhoff_03:00:00 (00:01:05:03:00:00), Dst: Beckhoff_04:00:00 (00:01:05:04:00:00)
3 Internet Protocol Version 4, Src: 10.1.5.3 (10.1.5.3), Dst: 10.1.5.4 (10.1.5.4)
4 Data (26 bytes)
5
6 0000  00 01 05 04 00 00 00 01 05 03 00 00 08 00 45 00   ..............E.
7 0010  00 2e 00 00 40 00 40 00 1c c8 0a 01 05 03 0a 01   ....@.@.........
8 0020  05 04 f6 f6 28 28 00 0c ff 83 df 17 32 09 4e d1   ....((......2.N.
9 0030  e7 cd d6 31 00 dc 8c 70 00 01 9c 89               ...1...p....
```

Listing 1: 60 bytes test frame send from port 3 of the Anritsu through an OpenFlow enabled switch to port 4 of the Anritsu

When the packet went through the OpenFlow switch and arrived at the Anritsu at port 4 it looks like showed in Listing 2, some how 8 bytes are added. A trailer is added (highlighted in dark-gray), this is unexpected behavior. After the trailer a standard Ethernet *Frame Check Sequence* (FCS) is added (highlighted in light-gray) this is expected behavior.

```
1  Frame 3: 68 bytes on wire (544 bits), 68 bytes captured (544 bits)
2  Ethernet II, Src: Beckhoff_03:00:00 (00:01:05:03:00:00), Dst: Beckhoff_04:00:00 (00:01:05:04:00:00)
3  Internet Protocol Version 4, Src: 10.1.5.3 (10.1.5.3), Dst: 10.1.5.4 (10.1.5.4)
4  Data (26 bytes)
5
6  0000  00 01 05 04 00 00 00 01 05 03 00 00 08 00 45 00   ..............E.
7  0010  00 2e 00 00 40 00 40 00 1c c8 0a 01 05 03 0a 01   ....@.@.........
8  0020  05 04 f6 f6 28 28 00 0c ff 83 df 17 32 09 4e d1   ....((......2.N.
9  0030  e7 cd d6 31 00 dc 8c 70 00 01 9c 89 30 6a da fd   ...1...p....0j..
10 0040  1c df 44 21                                       
```

Listing 2: 60 bytes test frame with 8 bytes added when recieved at port 4 of the Anritsu

We believe the OpenFlow switch perchance adds extra data for padding. We decided to send a bigger packet of 98 bytes in exactly the same setup. The Anritsu again received an unrecognizable test frame with 8 bytes added, 4 different bytes of trailer and a correct 4-byte FCS.

We decided to disconnect the switch from the NOX controller and install the flows manual in the switch using `dpctl`. in Listing 3 we observe a 60 byte send test packet from port 3 to port 4, in Listing 4 the same packet is captured but is 64 bytes long. This is the way the packet should arrive with only a FCS added. The test packet as send in Listing 3 is recognized by the Anritsu when it arrives at port 4. Unfortunately it makes no sense to run the performance overhead tests if we add the flows manually.

```
1  Frame 3: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
2  Ethernet II, Src: Beckhoff_03:00:00 (00:01:05:03:00:00), Dst: Beckhoff_04:00:00 (00:01:05:04:00:00)
3  Internet Protocol Version 4, Src: 10.1.5.3 (10.1.5.3), Dst: 10.1.5.4 (10.1.5.4)
4  Data (26 bytes)
5
6  0000  00 01 05 04 00 00 00 01 05 03 00 00 08 00 45 00   ..............E.
7  0010  00 2e 00 00 40 00 40 00 1c c8 0a 01 05 03 0a 01   ....@.@.........
8  0020  05 04 f6 f6 28 28 00 0c ff 83 df 17 32 09 4e d1   ....((......2.N.
9  0030  e7 cd 83 67 00 e6 df 30 00 01 9c 89               ...g...0....
```

Listing 3: 60 bytes testframe as send by the Anritsu with flows manualy installed

```
1  Frame 3: 64 bytes on wire (512 bits), 64 bytes captured (512 bits)
2  Ethernet II, Src: Beckhoff_03:00:00 (00:01:05:03:00:00), Dst: Beckhoff_04:00:00 (00:01:05:04:00:00)
3  Internet Protocol Version 4, Src: 10.1.5.3 (10.1.5.3), Dst: 10.1.5.4 (10.1.5.4)
4  Data (26 bytes)
5
6  0000  00 01 05 04 00 00 00 01 05 03 00 00 08 00 45 00   ..............E.
7  0010  00 2e 00 00 40 00 40 00 1c c8 0a 01 05 03 0a 01   ....@.@.........
8  0020  05 04 f6 f6 28 28 00 0c ff 83 df 17 32 09 4e d1   ....((......2.N.
9  0030  e7 cd 83 67 00 e6 df 30 00 01 9c 89 cc c7 55 7d   ...g...0......U
```

Listing 4: 64 bytes testframe as recieved at the Anritsu with flows manualy installed

When the controller receives the flow frame, it replies with a `OFPT_FLOW_MOD` message to install the flow in the switch. The `OFPT_PACKET_IN` message does not contain the actual frame, so it is unlikely that the controller is the cause of the malformed packet. We suspect that the 4 bytes trailer is added at the moment the frame is saved in the packet buffer on the switch or at the moment the packet is send out of the packet buffer. We did not investigate this problem any further.

Because of the problem described above it is not possible to test the performance overhead the controller adds in forwarding packets. Due to time limits we were not able to test the performance overhead on the NetFPGA hardware.

Controller Denial of Service

When a switch receives a packet from the network and there is no flow installed, the switch needs to ask the OpenFlow controller what to do with a specific packet. In Section 2.1.2 this process was already explained

and the latency of `packet_in` messages was tested.

Because the switching logic is removed from the switch and is placed on a separate controller, the controller becomes a critical and maybe vulnerable component. It may be flooded by `packet_in` messages from a switch with connected to a host sending out unknown floods at full line rate. If this means that other traffic isn't being exchanged anymore it could be considered a *Denial of Service* (DoS) attack towards the controller.

2.6.4 Test Set-up & Procedure

To generate unknown flows to send to the switch an Anritsu traffic generator is used. It is connected to the switch with two Ethernet interfaces. Both interfaces will send out frames with a randomized SA that will trigger the switch to send a `packet_in` message to the NOX controller.

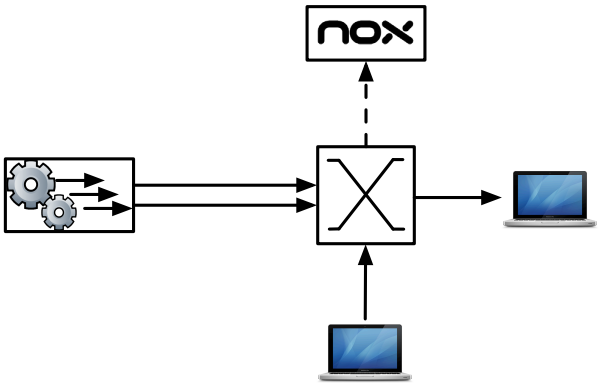

Figure 12: Test setup for unknown flow flood to NOX controller

While the random streams are being sent two hosts try to make a legitimate connection to each other. They have not previously exchanged traffic and thus the switch will also have to consult the controller for this traffic. This connection is monitored and the results documented, starting with the performance of `arping`s between the two hosts.

Because of the different OpenFlow implementation that is incompatible with the NOX contoller, the Open vSwitch implementation of the Pronto switches was not tested.

2.6.5 Results

As a baseline measurement the devices were tested without any unknown flows and letting the switch learn the SA of both hosts. The `arping` time measurements between the hosts are given in Table 9.

After this baseline, the NetFPGA implementation was tested with both Anritsu ports sending out full line rate random streams. This has a significant impact on the `arping` results, rising from 1.37 to 16.74 milliseconds, but still being able to exchange them.

	NetFPGA	Pica8 OpenFlow
No traffic	1.37 ms	8.50 ms
Random stream	16.74 ms	*N/A*

Table 9: `arping` results with and without a random traffic stream.

The Pronto switch with the Pica8 OpenFlow firmware did not perform that well. With no other traffic going over the switch it had an average ARP response time of 8.50 milliseconds. When the two random streams were started all ARP requests were dropped and setting up a connection was not possible. Stopping one of the Anritsu test ports did not have any effect, no ARP messages were received at the other end.

Installed flows per second	
16 switches	40,173 flows.
32 switches	40,607 flows.
64 switches	40,768 flows.
128 switches	41,132 flows.
256 switches	40,143 flows.
512 switches	37,625 flows.

Table 10: *cbench* result averages when testing NOX.

Although this seems a limitation on the switching firmware part, it is possible to determine many `packet_in` messages a controller can handle. 'Cbench' is a utility to emulate a number of switches, generates `packet_in` events and waits for flow modifications to be sent back from the controller. The earlier used OpenFlow controller NOX was tested using this tool, producing the results as seen in Table 10. The test consists of 105 one-second `packet_in` event streams for different numbers of emulated switches. The following command was used:

```
for i in 16 32 64 128 256 512; do  cbench -c 145.100.37.182 -p 6633 -s $i -l 105 -w 5; done
```

The results show a fairly consistent rise before the tool reaches 256 switches. At this point it has reached some sort of bottleneck in processing the `packet_in` events and resulting flow modification messages. A higher number of emulated switches than 512 unfortunately resulted in a TCP connection error towards the controller.

DoS attacks might pose a real problem in networks where connected users are able to send a lot of flows towards the controller. One way to mitigate this problem might be to implement some sort of 'implicit deny' rule in a separate, lastly checked flow table so that streams matching no exact or wildcard flow will be dropped immediately.

3 Conclusion

The tests performed during this research were planned to benchmark the available OpenFlow hardware at SARA. A summary of the results is given below, followed by some recommendations regarding researching and deploying OpenFlow in the future.

3.1 Results

In Section 1.2.1 the following main research question was posed *"How can the scalability of OpenFlow switches be explained by their hardware design?"*. The tests carried out to answer this question focussed primarily on the performance and OpenFlow functionalities of the hardware.

The available space in the flow tables of the Pronto switches was found to be fairly limited, with Open vSwitch differing from the OpenFlow implementation and implementing its own '`classfier`' table. The NetFPGA card on the other hand implemented both SRAM and TCAM memory for hash and linear lookups and yielded lower latency when tested using the RFC2544 procedure. Timing the installation of flows in a switch was done using a modified script of the `oftest` tool, which was incompatible with the OpenFlow implementation of the Open vSwitch software. The Pica8 OpenFlow software installed linear and hash flows within 4.33 and 2.03 milliseconds respectively, numbers that the NetFPGA could only beat on the linear flows, which it installed within 0.79 milliseconds. Installing hash flows in the off-chip SRAM took 6.13 milliseconds on average.

QoS features were nonexistent in pre-1.0.0 OpenFlow versions. The 1.0.0 version added only minimum rate queues to the specification, which unfortunately have not been implemented on all platforms. Open vSwitch for example has skipped this limited functionality and implements its own rate limiting functions. The NetFPGA build was tested but was found to actually rate limit the streams flowing through the queues, which could be attached to a port while in the documentation of the software it is said that the "Slicing feature is not supported". The Pica8 release notes specify that slicing should be supported but using the command line utilities queues could not be added to the OpenFlow stack. Overall, it is clear that the implementation of QoS services is yet to mature.

Mirroring ports using OpenFlow was found to have some effect on the maximum throughput of the switch. Although the results were not definite because of the lower throughput caused by the memory-to-memory test method of the IPerf UDP test tool, the results did suggest a measurable difference in throughput with and without mirroring.

Switching an OpenFlow path mid-stream has proven to be working flawlessly only on the NetFPGA implementation, which did not drop a single frame. On the Pica8 OpenFlow firmware on average 121.5 consecutive frames were dropped, resulting in 1 sequence error measured. The Open vSwitch implementation behaved a bit differently, causing an average number of sequence errors of 7,952 and sometimes even sending out more frames than the traffic tester sent out. One possible explanation might be that the ports reside in a mirroring state for a short period of time.

The performance overhead tests are based on measuring the latency of a first frame of an unknown flow that has to be learned by the controller. Again, Open vSwitch was not tested because of its incompatibility with the NOX controller. The Pica8 OpenFlow software was able to do so but while running the test, the test frames were not received the Anritsu in the way they were sent, the traffic tester was unable to recognize them and as such the latency measurements could not be performed. After further research the received frames appeared to have gained 4 bytes in transit to the receiving port, these bytes were not added when manual flows were installed to exchange traffic.

The Pronto switch with Pica8 software seems to be vulnerable to a DoS attack caused by the switch generating a lot of `packet_in` events after receiving unknown flows on one or more interfaces. Other connected hosts are unable to exchange ARP packets while such a stream is running because the packets are sent up to the controller, which is unable to process them. Using the NetFPGA card this only adds a longer reply time, but still making it possible to send and receive traffic.

Combining the results to form an answer to the research question it becomes apparent that although the most basic forwarding features are available in the implementations tested, scalability is a characteristic that cannot yet be added to the OpenFlow feature set of the tested hardware. While the performance scalability could be improved by changing their hardware design, one of the more significant bottlenecks seems to be the interoperability between OpenFlow firmware, their hardware and the software used to control them.

3.2 Recommendations

Testing hardware performance of switches and protocols in development makes it difficult to benchmark them. Implementations vary per platform and tools aren't developed in the same pace. The Open vSwitch implementation for example was not compatible with any of the benchmarking tools or controller software that was available at the time of writing. This incompatibility was caused by Open vSwitch implementing features from the 1.1.0 and even 1.2 specifications while it still claims to be 1.0.0 when polled for a feature request.

At the end of this project (February 2012) a release of the OpenFlow 1.2 specification was imminent. This new version will not be backwards compatible with 1.0.0 or 1.1.0. Two months later in April 2012, the next 1.3 version is already scheduled to be released.[11] This fast-paced release cycle might be beneficial for the maturity of the protocol but this also means that all tools and hardware implementations will have to be updated at the same pace.

Until the releases of the OpenFlow specification become more consistent and stable it is unlikely that a common benchmarking tool will be available. When this will happen depends on when vendors start implementing a stable version of the specification and developers update their software for this version.

[11] https://www.opennetworking.org/1dot2-and-pending

A Sources

```
#!/bin/sh
incr=1025
max=5026

while [ $incr -le $max ] ; do
        ovs-ofctl --strict add-flow br0 in_port=1,idle_timeout=0,dl_vlan=1,dl_vlan_pcp=2,dl_src=00:00:00:00:00:
             05,dl_dst=00:00:00:00:00:05,dl_type=0x0800,nw_src=192.168.1.1,nw_dst=192.168.1.2,nw_proto=6,nw_tos=
             5,tp_src=22,tp_dst=${incr},actions=output:3
        if [ $? -ne 0 ] ; then
                exit 1
        fi
        incr=$(expr $incr + 1)
        echo $incr
done
```

Listing 5: Shell script to install flows on a switch.

```
...

    start = time.time()
    while True:
        request.match.nw_src += 1
        rv = obj.controller.message_send(request)
#        do_barrier(obj.controller)
        flow_count += 1
        if flow_count % count_check == 0:
            response, pkt = obj.controller.transact(tstats, timeout=2)
            obj.assertTrue(response is not None, "Get tab stats failed")
            caps_logger.info(response.show())
            if table_idx == -1:  # Accumulate for all tables
                active_flows = 0
                for stats in response.stats:
                    active_flows += stats.active_count
            else: # Table index to use specified in config
                active_flows = response.stats[table_idx].active_count
            if active_flows != flow_count:
                stop = time.time()
                break

...
```

Listing 6: Python script to install flows and measure the time it takes.

B Acronyms

ARP *Address Resolution Protocol*

API *Application Programming Interface*

DA *Destination Address*

DoS *Denial of Service*

FCS *Frame Check Sequence*

FPGA *Field Programmable Gate Array*

ICMP *Internet Control Message Protocol*

IEEE *Institute of Electrical and Electronics Engineers*

IO *Input/Output*

IP *Internet Protocol*

ITU-T *International Telecommunication Union - Technology*

LAN *Local Area Network*

MAC *Media Access Control*

ONF *Open Networking Foundation*

OS *Operating System*

PCI *Peripheral Component Interconnect*

OSI *Open System Interconnect*

QoS *Quality Of Service*

RAM *Random Access Memory*

SA *Source Address*

SDN *Software Defined Networking*

SRAM *Static* RAM

TCAM *Ternary Content-Addressable Memory*

TCP *Transmission Control Protocol*

ToR *Top of Rack*

UDP *User Datagram Protocol*

VLAN *Virtual* LAN

C Bibliography

[1] Brandon Heller et al., "OpenFlow Switch Specification – Version 1.0.0," December 2009. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[2] R. van der Pol, "Ethernet OAM enabled OpenFlow Controller." https://noc.sara.nl/nrg/presentations/SC11-SRS-8021ag.pdf.

[3] M. P. Mateo, "OpenFlow Switching Performance," July 2009. http://www.openflow.org/downloads/technicalreports/MS_Thesis_Polito_2009_Manuel_Palacin_OpenFlow.pdf.

[4] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, "OpenFlow Switching: Data Plane Performance," 2010. http://www.telematica.polito.it/~bianco/Papers_pdf/2010/icc_openflow.pdf.

[5] D. Talayco, "Indigo - OpenFlow for Hardware Switches." http://www.openflowhub.org/display/Indigo/Indigo+-+OpenFlow+for+Hardware+Switches.

[6] J. Naous, D. Erickson, and T. Yabe, "OpenFlow NetFPGA 1.0.0." http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/OpenFlowNetFPGA100#Description.

[7] Prontosys, "Pronto 3290 Specification." http://www.prontosys.net/pronto3290.htm.

[8] Charlie Schluting, "On Your Network: What the Heck is a TCAM?," August 2005. http://www.enterprisenetworkingplanet.com/netsysm/article.php/3527301/On-Your-Network-What-the-Heck-is-a-TCAM.htm.

[9] Brad Hedlund, "On data center scale, OpenFlow, and SDN," April 2011. http://bradhedlund.com/2011/04/21/data-center-scale-openflow-sdn/.

[10] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices." http://www.ietf.org/rfc/rfc2544.

[11] B. Pfaff, "Revise OpenFlow 1.0 ofp-match normalization." http://openvswitch.org/pipermail/dev/2011-May/008513.html.

[12] T. Yabe, "OpenFlow NetFPGA 1.0.0." http://www.openflow.org/wk/index.php/OpenFlowNetFPGA1_0_0.

[13] D. Talayco, "OpenFlowHub – Question about flow matching implemenation." http://forums.openflowhub.org/showthread.php?18-Question-about-flow-matching-implemenation.

[14] O. vSwitch Development Team, "How to Port Open vSwitch to New Software or Hardware." http://openvswitch.org/cgi-bin/gitweb.cgi?p=openvswitch;a=blob;f=PORTING;hb=023e1e0a4d0a9f6937d9a2c66ba30d502029c4df.

[15] *International Telecommunication Union - Technology*, "OAM functions and mechanisms for Ethernet based networks," 2008. http://www.itu.int/rec/T-REC-Y.1731/en.

Acknowledgements

Thanks to Ronald van der Pol at SARA *who even when he is on the other side of the world is a great supervisor and provided us with a lot of feedback on our report*

We would also like to thank AMS-IX *for providing us with an Anritsu traffic generator and their feedback on our report and presentation.*

Lastly we would like to thank Pieter Lexis, Jeffrey Sman and Freek Dijkstra for reviewing and proofreading our report.