D5.4 Initial Benchmarking Documentation

Dissemination level	PU
Version	1.0
Due date	31.01.2015
Version date	03.04.2015

This project is co-funded
by the European Union

Document information

Authors

Editor: INTEL

Contributors:

Hagen Woesner (BISDN), Gergely Pongracz and the Cloud Research group at Ericsson Hungary (ETH), David Verbeiren (Intel)

Reviewers:

Wolfgang John (EAB), Wouter Tavernier (iMinds)

Coordinator

Dr. András Császár

Ericsson Magyarország Kommunikációs Rendszerek Kft. (ETH) AB

Email: andras.csaszar@ericsson.com

Project funding

7th Framework Programme

FP7-ICT-2013-11

Collaborative project

Grant Agreement No. 619609

Legal Disclaimer

Revision and history chart

Version	Date	Comment
0.1	14.10.2014	Initial version
0.2	24.11.2014	Updated with lwAFTR measurements
0.3	11.02.2015	Added Hypervisor and BNG benchmarks
0.4	02.03.2015	Improved introduction and restructured sections
0.5	09.03.2015	Added measurements with multiple packet sizes for Xeon tests.
0.6	16.03.2015	Addressed most comments from first review.
0.7	31.03.2015	Updated after WP5 internal review; latest CAROS appliances results.
0.8	01.04.2015	Alignment on throughput reporting and final input (Xen w/ Netmap)
1.0	03.04.2015	Final review outside WP5

Table of contents

1 Introduction

The purpose of the UNIFY work package "Universal Node Architecture and Evaluation" is to investigate how and to which extent the paradigm that has emerged in the IT world, where standard hardware is used to deploy a large variety of services, could translate into communications networks and what benefits and costs it would bring. Previous deliverables of this work package have addressed the description of the "Universal Node functional specification and use case requirements on data plane" in D5.1 [1], the "API and Universal Node software architecture" in D5.2 [2] and the "Prototype description and implementation plan" in D5.3 [3].

This deliverable, "Initial Benchmarking Documentation", is the first in a series that will present the results of the Universal Node (UN) performance characterization and development activities. The work is still at a very early stage of this task and most of the results that are presented in this deliverable are about getting baseline measurements. However, these baseline measurements will be useful in future work in the project, when evaluating the performance improvements brought by a new implementation or the impact of additional functionality, as input data for a UN local orchestrator, or when doing bottleneck analysis of specific use cases.

Sections 2 and 3 present results that characterize the performance of the major individual building blocks of the UN. In Section 2 the Virtual Switching Engine is investigated based on different realisations, such as xDPd [4] OpenFlow Soft Switch on the CAROS appliances and on Intel Xeon based platforms, and an OpenFlow Soft Switch with Run-Time Code Generation. In Section 3 two hypervisors are compared for the case where DPDK-based packet processing applications are executed within the Virtual Machine. The evaluated hypervisors are Xen [5] and QEMU-KVM [6].

In section 4, the first complete test setups are introduced where actual functions execute on top of the basic building blocks. Initial results are provided for a dummy function meant to serve as baseline measurement and for an IPv4 over IPv6 tunnel concentrator (lwAFTR) function. Those functions were implemented based on DPDK and executed in various environments. Finally the latest results of a broadband network gateway (BNG) implementation running bare-metal are provided as baseline for the work that is being done adapting this workload to run as a set of VNFs executing on the UN.

Note: Unless stated otherwise, the forwarding throughput values reported in this deliverable represent the rate at which packets go through the described system. Packets are only counted once, i.e. at the ingress or at the egress but not at both. Unless stated otherwise, the values represent the total throughput for the number of ports and CPU cores active in the specific test.

2 vSwitch Performance

This section presents initial performance evaluations of two vSwitch implementations that can be considered for the UN (in addition to OVS which will be included in future tests). The first one, xDPd with DPDK-based I/O, was tested on a range of x86 based platforms from an Intel® Atom™ based appliance to an Intel® Xeon™ based platform. A second implementation, with Run-Time Code Generation optimization, focuses on improving the overall performance by using template-based code generation.

It should be noted that, at this stage, the performed evaluations targeted specific goals and, as a result, often do not allow direct comparisons between them.

2.1 xDPd OpenFlow Soft Switch

In this testing, the platforms are running the xDPd soft switch configured to use DPDK for I/O acceleration into and from the pipeline. The pipeline is however the plain ANSI-C pipeline of xDPd that doesn't use DPDK libraries for matching or hashing. The use of these libraries would certainly lead to a further speedup at the price of a loss of portability across platforms (which was the first design goal of xDPd).

2.1.1 Results on the CAROS Appliances

The first step in measuring behaviour of a soft switch in a more practical scenario was undertaken for two hardware appliances that are to be found at the lower end of the commodity hardware market. BISDN uses two types of hardware appliances that are originally made by Lanner, Inc., and typically used as firewall appliance.

*Figure 2-1 - CAROS 75101 appliance, equipped with Intel® Atom™ D525 and 6 GbE interfaces. CAROS 87110 has an additional 2*10GbE slot on the right and a more powerful CPU (Intel® Core™ i5).*

The boxes are based on Intel® x86 platforms with 6 GbE interfaces on-board. The low-end CAROS 75101 is based on an Intel® Atom™ D525 dual-core 1,8 GHz processor, with 2GB DDR2 RAM and 4GB Flash Memory, while the more powerful CAROS 87110 features an Intel® Core™ i5-3550 quad-core 3,3 GHz processor ("Ivy Bridge"), 8GB DDR3 RAM installed, and two 10GbE interfaces (Intel® 82599) that are attached to the board via PCIe 2.0.

2.1.1.1 Test Setup

First measurements were done with an IXIA test equipment that is situated at Deutsche Telekom Innovation Labs in Winterfeldtstraße, Berlin. BISDN offices recently were connected to DT Labs by fibre cable and so two of the fibres were used to directly attach to 10G ports via SR SFP+ interfaces. For the lower bitrate tests (up to 1GbE line rate) a Spirent TestCenter Virtual Machine was used that is part of the OFELIA testbed. Nevertheless, as will be shown in the test results for the pure L2 forwarding, the Spirent VM is becoming unreliable beyond 1.2 Mpps and therefore a third traffic generator, DPDK-Pktgen was used.

Software Versions:

- xDPd: devel-0.6 branch of January 2015 (git://github.com/bisdn/xdpd;branch=devel-0.6), built with DPDK 1.7.1
- DPDK-Pktgen: v. 2.7.7 (with DPDK 1.7.1)

xDPd was also connected to an external OpenFlow 1.3 controller (Ryu [7]) that ran an unchanged simple_switch_13.py plugin that turns the soft switch into a simple learning switch. Figure 2.2 sketches the setup of the measurements.

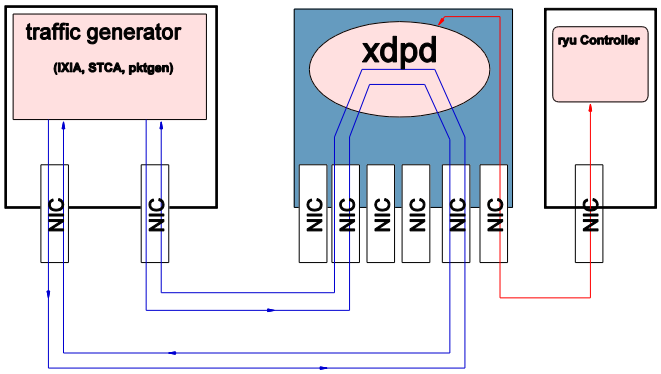

Figure 2.2 - Measurement setup for L2 forwarding rate measurements. Traffic generators varied based on the inputs to the DUTs (10GbE required optical inputs).

Traffic originates from both ports of the traffic generator and traverses the system in both directions between the two connected ports.

2.1.1.2 Forwarding Throughput

Figure 2.3 shows the results of the plain L2 throughput measurements for various packet sizes. The forwarding rate is shown, in millions of frames per second, as a function of the offered load. On the left, for the Atom-based appliance, the line rate (2 ports) is 2 Gbps (2.98 Mpps for 64 byte packets), whereas it is 20 Gbps for the Core i5-based system on the right.

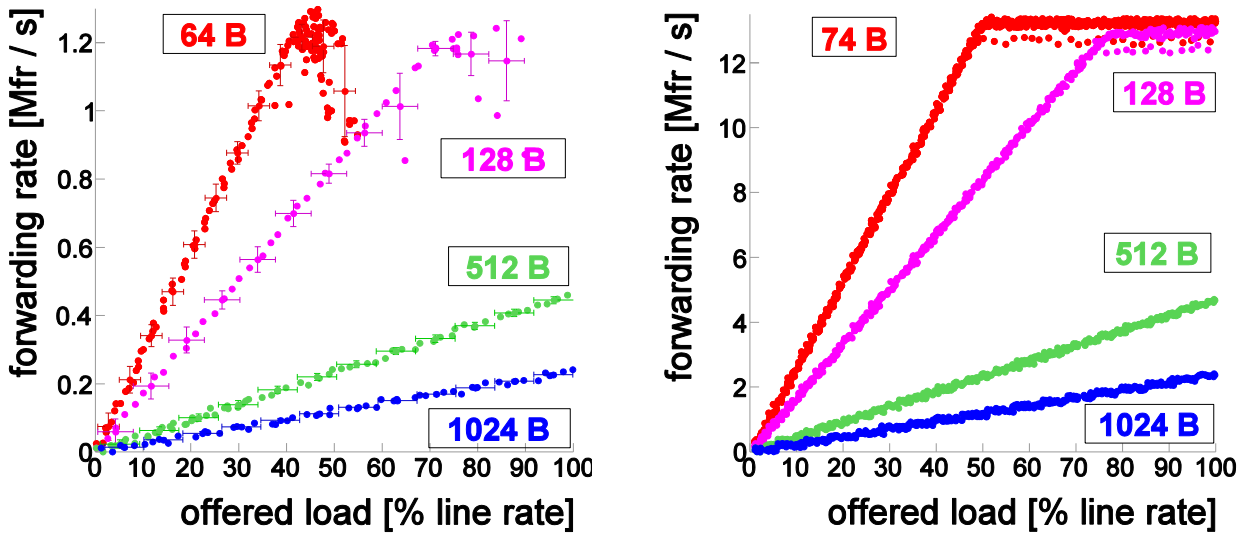

Figure 2.3 – L2 throughput results (in Millions of packets/frames per seconds) for CAROS boxes, left the Atom-based, right the Core-i5-based machine.

The instability in the left graph is most likely caused by the traffic generation in the Spirent TestCenter Anywhere (virtual machine).

In both cases, xDPd-dpdk used 3 cores, one for management and one processing core per port.

Results show that DPDK indeed increases throughput, even for low-end devices equipped with Atom CPUs. Other measurements without DPDK (using MMAP) yield about 10% of the throughput obtained with DPDK.

The Intel Core-i5 based machine shows a performance of about 13 Mpps for all frame sizes (up to the limitation of the line rate). This indicates that the actual bottleneck is the header processing in the pipeline.

Note that the minimum frame sizes are different in the two sub-figures in Figure 2.3. While IXIA in this setting generates 74 byte frames (when configured to do 64 byte), the Spirent Virtual TestCenter actually creates 78 bytes for a nominal 64 bytes. Pktgen was indeed generating 64 bytes on the wire.

2.1.1.3 Forwarding Latency

For the CAROS 87110 which was measured remotely via an IXIA, the latency values start from 18μs. It has to be noted that the round-trip time between the testing device and the device under test was measured as 15 μs, indicating a fibre length of around 1.5 km between the two buildings (Google maps shows the shortest path to be 1.1 km).

This indicates a total latency of around 3 μs, which is surely acceptable for a soft switch, as it is obviously easily dominated by the physical transmission delay.

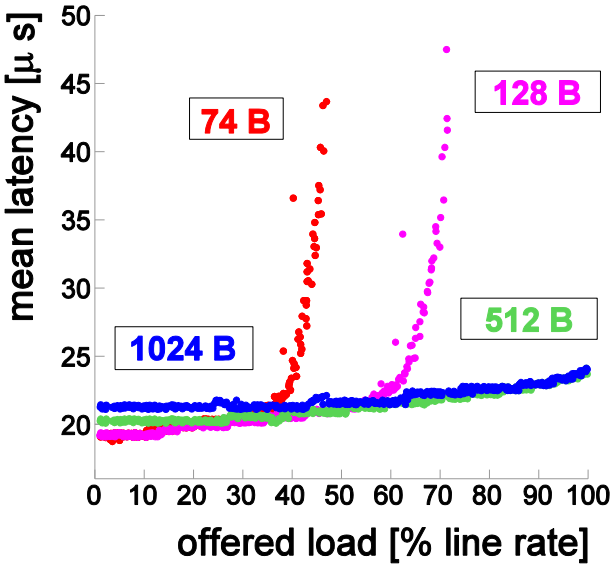

Figure 2.4 – Latency of xDPd-dpdk running in CAROS 87110, floor of 15 us needs to be taken into account for fiber transmission.

2.1.2 Results on Intel Xeon based Platform

Tests of the xDPd soft switch running on a server class CPU were also performed.

2.1.2.1 Test Setup

The test setup was very similar to that described in section 2.1.1, but using OpenDayLight as OF controller and only using DPDK-Pktgen as load generator.

The system under test consisted of a dual socket server with the following relevant components and settings:

- Intel® Xeon® E5-2697 v2 processors (2.7 GHz) of the "Ivy Bridge" generation
- 10Gb "Niantic" (82599) network interfaces
- CPU frequency was set at constant 2.7 GHz (Turbo, P-States and C-states disabled)

Simple flow rules, matching on input port only, to forward traffic between two ports in both directions were installed in the switch using the OpenDayLight controller.

2.1.2.2 Results

The maximum forwarding throughput between two ports was measured while varying the packet size from 64 to 512 bytes. The measurement was repeated for different numbers of CPU cores handling the packet forwarding: 1, 2 and 3 cores (different CORE_MASK values in the xDPd DPDK backend driver). The results are shown on Figure 2.5 where the forwarding throughput is the total for both ports (packets are forwarded from port 1 to port 2 and vice versa, hence a total line rate of 20 Gbps or 29.76 Mpps).

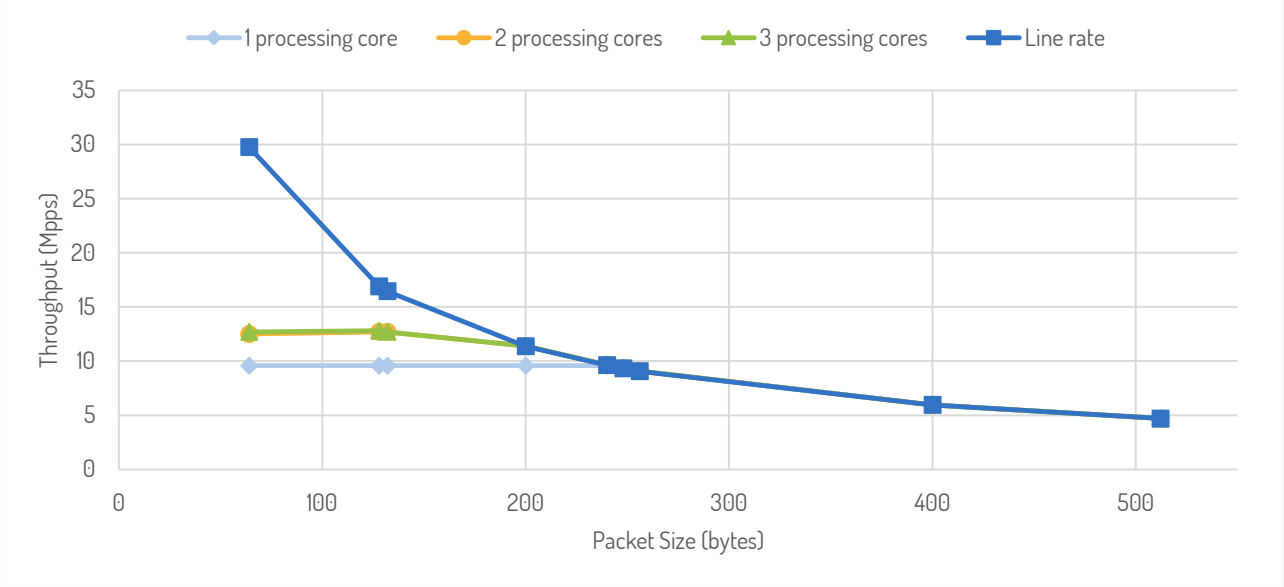

Figure 2.5 - xDPd Forwarding Performance on 2 x 10GbE ports on Intel Xeon E5 2697 v2

It should be noted that, for each packet size where the maximum throughput is lower than the line rate, the forwarding throughput remained at its peak value when a higher load was offered to the system.

With a single core handling the bi-directional traffic between two ports, line rate throughput was achieved for packet sizes of around 256 bytes and higher. By adding a second CPU core, line rate could be achieved starting at 200 bytes. The addition of a third core did not bring any further improvement, which was to be expected given the port to core assignment scheme currently implemented in xDPd (round-robin).

Similarly, the maximum throughput of 12.7 Mpps was achieved with 2 processing cores. The single core configuration peaked at 9.57 Mpps.

This scaling is currently limited to one core per port, and additional cores are simply not used. A possible improvement would be to configure multiple queues per port and use the distribution function of the NIC (Receive-

Side Scaling) or use an architecture where the I/O cores act as load-balancers for a pool of switching cores. However, such improvement may not benefit all use cases as the bottleneck may be found elsewhere in the system when more complete setups are used, with VNFs running on top of the soft switch, as is the case in the configurations studied in section 4. Additionally, further investigation is needed to understand why the throughput only improved by around 30% when adding a second core to handle the traffic of both ports.

2.2 OpenFlow Soft Switch with Run-Time Code Generation

Since one of the main – and probably the most performance-sensitive – part of the Universal Node is the virtual switch, it is imperative that it is optimized as much as possible. Typically the bottleneck of the switch is its packet processing capacity where the problem gets worse with decreasing packet size. Also – unlike in case of a datacenter-focused solution where the vSwitch is just used to distribute packets to the correct VMs and the network functions are running inside the VMs – the UN can host several VNFs inside its virtual switch, which means that the switch itself runs VNF functionality so it has to handle rather complex pipelines and a wide variety of OpenFlow actions.

In this section two optimization techniques are presented that can increase a switch's capacity especially when it has to deal with small packets or complex instruction/action set.

An OpenFlow (OF) switch that uses run-time code generation[1] and automatic lookup algorithm selector was developed and measured against a "normal" (e.g. generic if-then-case and function call based action handling) switch implementation, in a router configuration. It should be noted that the processing for this case is significantly more advanced than the simple forwarding done in the tests of section 2.1 since it includes tables lookups based on L3 header information and header modifications.

2.2.1 Run-Time Code Generation (RTCG)

Using template-based code generation, the switch can dynamically link together only the code pieces that are needed by the OpenFlow rules received. This works best for action handling but in some cases it can also be used in lookups and inter-table navigation. This approach fits OpenFlow quite well because its rules use a relatively small, well defined set of match functions and actions, which can be precompiled as templates.

2.2.2 Automatic lookup algorithm selection

Although RTCG can be used for speeding up some lookups, the most significant gain for the match functions is coming from the optimization of lookup algorithms. The switch analyses the match fields in each flow table, and uses a special lookup method if it finds some patterns in the incoming match rules. It can use hash if the match field is a direct match for the same field among all the rules or it can detect longest-prefix match rules as well. On the other hand if the number of rules is very small but the rules are very different, it will use an RTCG generated code

[1] This was inspired by a technique seen in the UPX compression utility (http://upx.sourceforge.net/)

for the lookup. The IPv4 lookup uses the DPDK provided functionality where a trie with extreme level compression (see e.g. [8]) is used, resulting in at most two memory accesses per lookup.

2.2.3 Mitigating the Cost of Code Generation

The overhead of the code generation step and the possible problem of instruction cache overload can be mitigated by using a generic implementation when rules are first created, and only optimize the matches and instructions that are used frequently with respect to some counters or statistics.

2.2.4 Measurement Setup

The measurements focused on the usage of the OF switch prototype in a router configuration. They include both synthetic scenarios (e.g. smallest packet size) as well as very realistic ones (real forwarding information base (FIB), traffic based on real traces).

The physical setup is first described, followed by the OpenFlow pipeline configuration, the content of the forwarding table (FIB) and finally the different test traffic types applied to the system.

2.2.4.1 Hardware Configuration

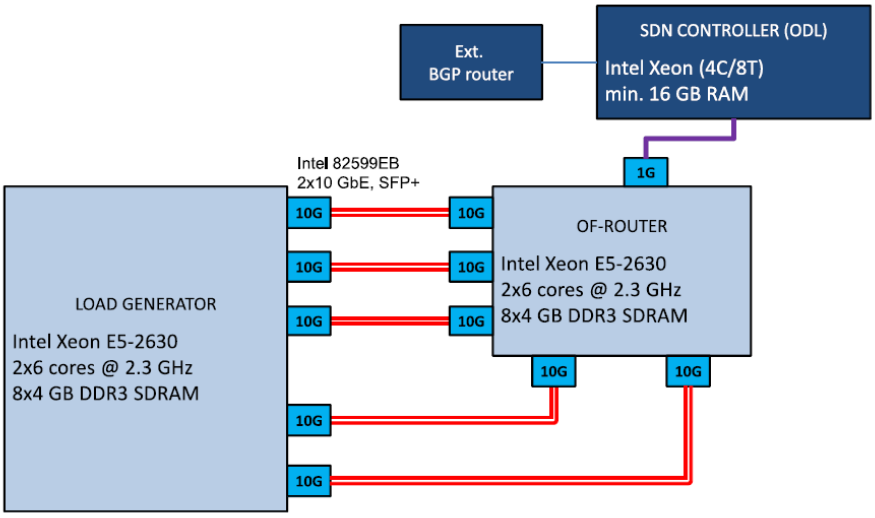

Figure 2.6 - Network Setup

The setup, depicted on Figure 2.6, consisted of two identical systems connected over 10GbE links, one system acting as load generator and the other one running the OF Router prototype. The OF router system was also connected to an SDN controller but this is not directly used in the tests described here.

The two systems were Supermicro workstations with the following configuration:

- Intel Xeon E5-2630 (Sandy Bridge) processors, each having 6 cores running at 2.3 GHz(32kB + 32kB L1 cache, 256kB L2 cache, 15 MB L3 cache)
- Fully populated DRAM banks to maximize the usage of memory channels
- 5 x Intel 82599EB (2x10GbE) based NICs, altogether 10 x 10GbE ports

2.2.4.2 Software Configuration

DPDK-Pktgen was used on the load generator, generating 10 Gbps traffic on each port. That means that the system under test was dealing with 100 Gbps of incoming traffic during most of the tests.

The RTCG OpenFlow prototype code was running on 11 cores of the OF router system: 10 cores were polling the selected Ethernet port in busy-loop, while the last core was listening for the control channel.

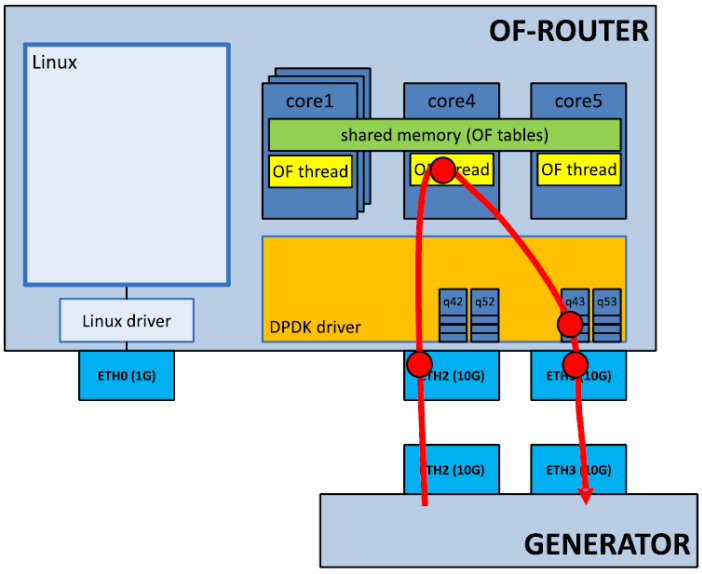

Figure 2.7 - OF Router Software Configuration

The OpenFlow pipeline setup can be seen in Figure 2-8. Note that the figure describes a virtual routing and forwarding (VRF) scenario where, besides the destination IP address, the FIB also takes VLAN value into account. The measurements concentrated on a normal router scenario where this additional data is not used.

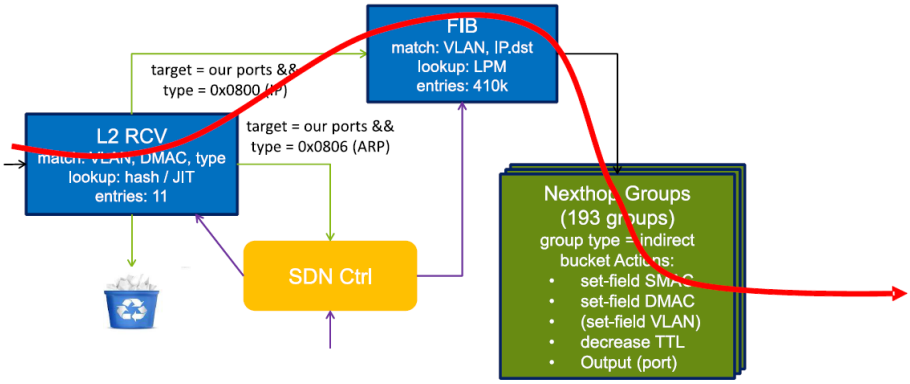

Figure 2-8 - OpenFlow pipeline setup for the SDN router

To simulate a real router and also in order not to waste memory space, the OpenFlow pipeline was designed to contain 2 tables and 1 group. The first (L2 rcv) table contains only a few L2 (Ethernet) rules, filtering out the ARP messages and dropping the non-IP packets. The second table (L3 (FIB)) is where routing is done: from the destination IP address we find the proper next hop here in the form of a group. In theory the second (FIB) table could also contain nexthop information for each and every prefixes, but in that case there would be a huge amount of duplicated information, which would waste memory and would also make nexthop updates really hard.

2.2.4.3 FIB Setup

In the measurements the system was configured using a real forwarding information base (FIB) table that was downloaded from an access router in the Hungarian Internet backbone (hbone). The prefix length distribution was quite uneven, more than 50% of the prefixes was /24 prefix, and the vast majority was between /16 and /24.

The statistics can be seen in Figure 2.9. The FIB contains more than 400k entries (410454) and the number of next hops is 194.

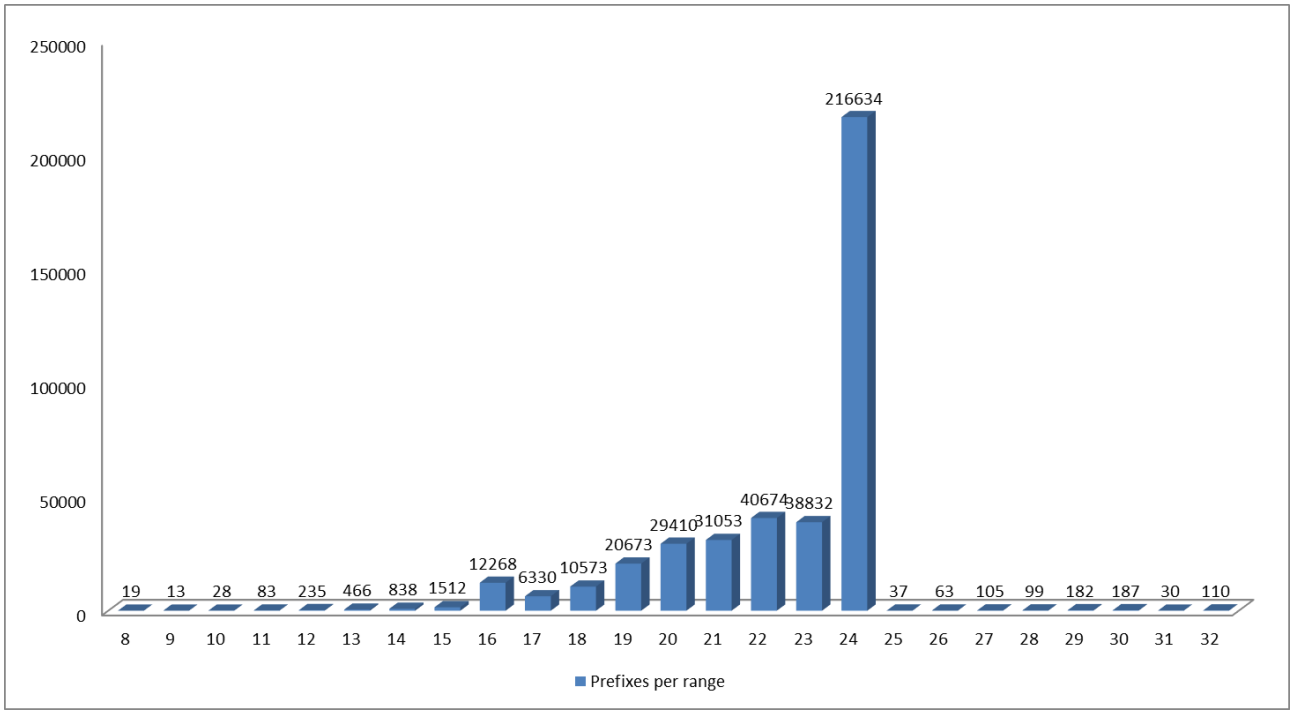

Figure 2.9 - The structure of the FIB: prefix length distribution

2.2.4.4 Test Traffic

The following different types of test traffic were used:

- "Single flow": traffic consisting of TCP packets of a fixed length. Packet lengths from 64 bytes to 512 bytes were tested: further increase made no sense since the bandwidth of the physical links was already the limiting factor at this packet size.
- "Realistic": a synthetic trace constructed to emulate a captured, full payload packet trace from a mobile operator
    - The packet size distribution emulates the original trace with small, medium and large sized packets
    - 1-1 mapping to IP flows of the original trace, preserving the number of different destinations as in the original trace, as well as the number of packets per flow destination
    - Contrary to the situation in the original trace, the destination is selected with an algorithm that makes the load evenly distributed among the egress interfaces
- "Almost Worst Case": another constructed trace put together to act as a(n almost) worst case scenario
    - The packet size is always set to small (64B)

- One original IP flow is mapped to one packet (only the first packet was taken), resulting in destinations that usually change from packet to packet
- The destination is selected with an algorithm that makes the load evenly distributed among the egress interfaces

2.2.5 Results

Figure 2.10 shows the performance of the OF pipeline in case of a single port handling the "Single flow" test traffic transmitted by the test generator. In this case the receiver port only receives traffic, while the nexthop port of the router only transmits traffic.

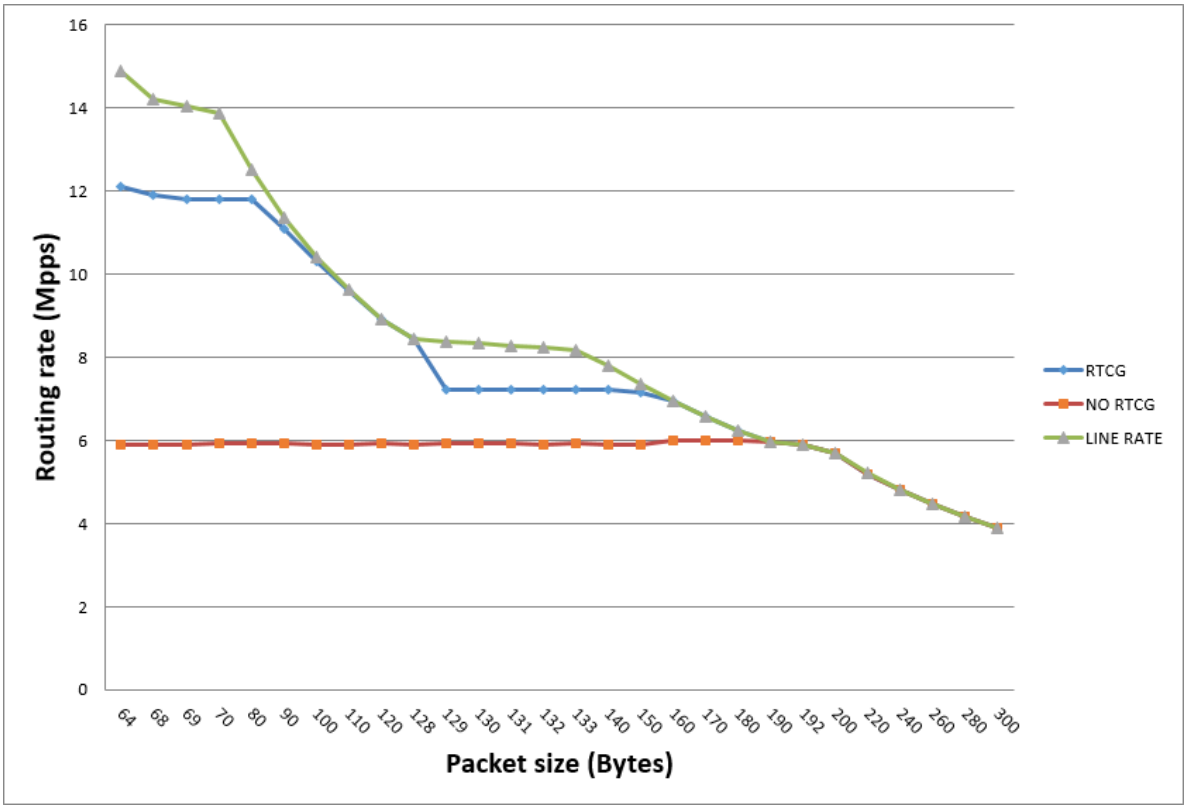

Figure 2.10 - OF pipeline packet processing performance, single port

The achieved throughput is shown in millions of packets per second (Mpps) as a function of the packet size, with (blue curve) and without (red curve) the run-time code generation optimization. The theoretical maximum rate is also shown (green curve).

The maximum rate (12 Mpps) achieves around 80% of the theoretical maximum while line rate is reached at around 100 byte packet size. An interesting phenomenon, a sudden drop of the forwarding rate, can be seen between 128 and 129 bytes. This is discussed later.

Figure 2.11 shows the same "Single flow" measurements for the full capacity of the SDN router, when all 10 ports are used and loaded with 10 Gbps full duplex traffic (traffic received from the load generator on the 10 ports leaves the router on the same 10 ports).

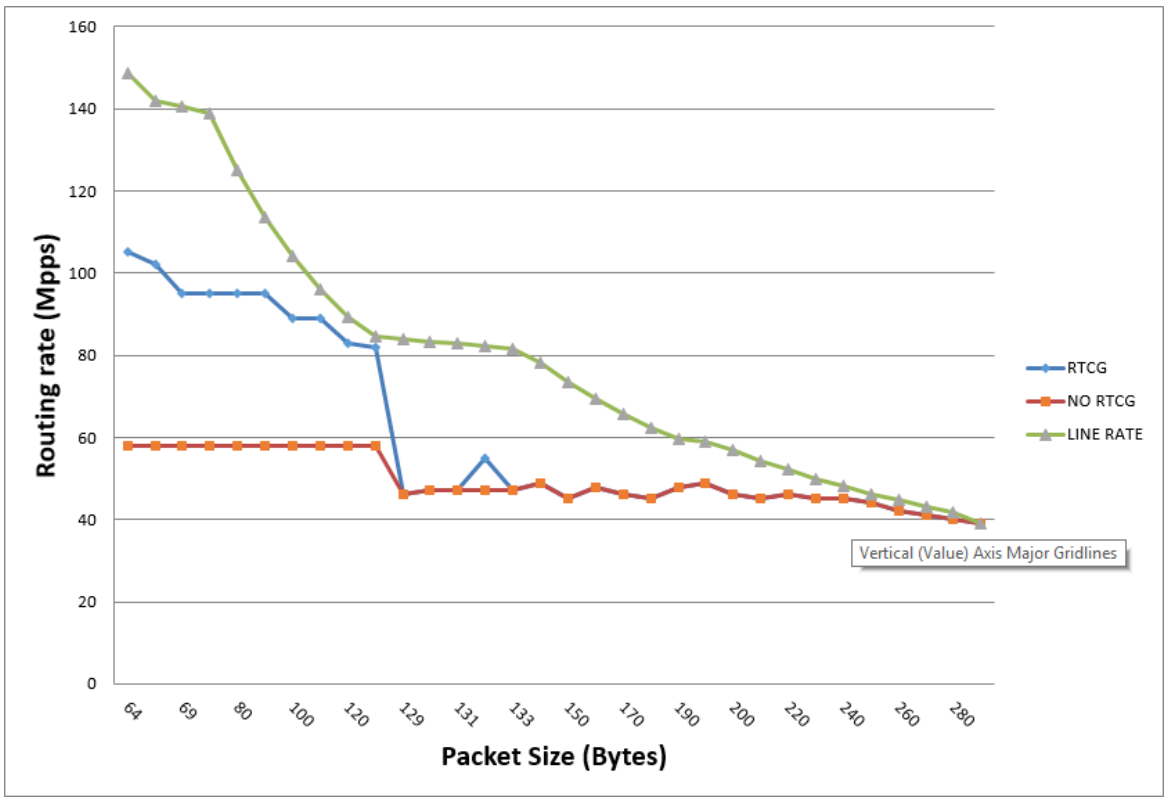

Figure 2.11 - OF pipeline packet processing performance, 10 ports

Comparing with the previous case, it can be seen that the throughput did not scale perfectly with the number of ports, despite the 1-to-1 mapping of ports to processor cores. There is a performance drop of around 10%. Another observation is that the drop between 128 and 129 byte packet length, which was already present in the single port case, is much more significant, almost 50% in the fully-loaded case.

The scalability issue is in fact an effect of the test setup and a network interface card limitation. When all ports are used, despite the generators only generating "transmit" direction traffic, all ports simultaneously receive and transmit at 10 Gbps. Bi-directional line rate on both ports cannot be achieved with the Intel 82599 2x10Gb Ethernet controller because of the bandwidth limitation of the PCIe Gen2 interface to the host. Even though the theoretical PCIe bandwidth is sufficient for 20 Gbps, there is inherent overhead in the protocol used over the interface and also additional data transfers between the controller and the host memory to maintain RX and TX descriptors. As a result, only around 80% of the bi-directional line rate can be achieved for 64 byte packets.

We have no definitive explanation for the drop of performance when going from 128 to 129 bytes packet sizes, besides that it is most probably not related to the OpenFlow implementation, since the exact same behaviour could be seen with other DPDK-based examples, like l2fwd.

Using PCM[2] we observed that the reported instructions per CPU cycle drops significantly between the two packet sizes, and that the L2 and L3 cache misses increased significantly, during the measurement with our OF switch running on 10 cores. However, in later measurements with a newer, 40 GbE dual-port Intel NIC (x710), neither the 10% scalability loss nor the 128-129 byte drop were present. This clearly shows that with the 82599 NICs we hit the PCIe bus limitation and that the 128-129 byte drop was likely due to a driver issue.

Table 1 shows the summary of all the measurements performed.

Traffic type	RTCG disabled (Mpps)	RTCG enabled (Mpps)	RTCG disabled (Gbps)	RTCG enabled (Gbps)
"Single Flow" - 64b	65	110 (+69%)	44	75
"Single Flow" - 128b	65	82 (+26%)	77	97
"Single Flow" - 256b	44	44	100	100
"Single Flow" - 512b	23	23	100	100
"Realistic"	29	29	100	100
"Almost Worst Case"	45	77 (+71%)	32	55

Table 1 - OF Router Results Summary

As one can see the SDN router performs on line rate in almost every use case. Real-life use cases were handled easily even without using the RTCG-Optimized pipeline. In these cases the reason for using the new pipeline is to gain some room for more complex rule sets and/or decrease the number of CPU cores allocated to switching on the platform thus making room for more VNFs. Another reason is to handle higher capacity network cards, such as 40GbE or 100GbE.

2.3 Conclusions

Early evaluations of soft switch implementations presented in this section confirmed that, using the software acceleration techniques provided by DPDK, commodity hardware can reach the type of performance targets set in D5.1. They also showed the potential benefits of using Run-Time Code Generation and automatic lockup algorithm

[2] a tool that visualizes the performance counters of the Intel CPU

selection to improve the performance further, especially for small packet sizes. These evaluations were however limited to the soft switches themselves. Tests with VNFs on the data path will be presented in the following sections.

3 Hypervisor Performance

Alongside the vSwitch, the hypervisor is another key technology to support flexible deployment of VNFs on the Universal Node. In this section, we compare the Xen and QEMU-KVM hypervisors for the case where DPDK-based packet processing applications are executed within the Virtual Machine. Both technologies and how they differ were introduced in [2] which also introduced the DPDK libraries.

3.1 Experimental Setup

The measurements are performed using two physical machines, one of which is used as a host machine for guest VMs. A virtual switch (or bridge, depending on the scenario) is running on the host machine, which connects the VMs with each other and with the other physical machine.

3.1.1 Physical Systems

Below are the most important details about the physical systems used for the experiments.

Hardware:

- Intel Xeon CPU E5-1650 v2 @ 3.50GHz (6 cores, 12 threads)
- 32 GiB DDR3 RAM
- Intel 82599ES 10-Gigabit NIC (Dual Port)

BIOS settings:

- Intel VT-x and VT-d enabled
- CPU Power and Performance Policy set to Balanced Performance

3.1.2 Examined virtual bridges/switches

The following virtual bridging/switching solutions were used:

- Open vSwitch 2.0.1 (OVS) [9]
- Intel DPDK vSwitch 0.10.0-30 (OVDK) [10]

3.1.3 Setup for Xen measurements

The Xen measurements are performed using two guest domains (VMs) (G1, G2) running on the host machine. The bridge/switch (SW) connecting the virtual interfaces and the host's physical ports is operated in the Xen control domain (dom0). The other physical machine (C) is used to generate traffic for scenarios involving external traffic. The setup is illustrated in Figure 3.1.

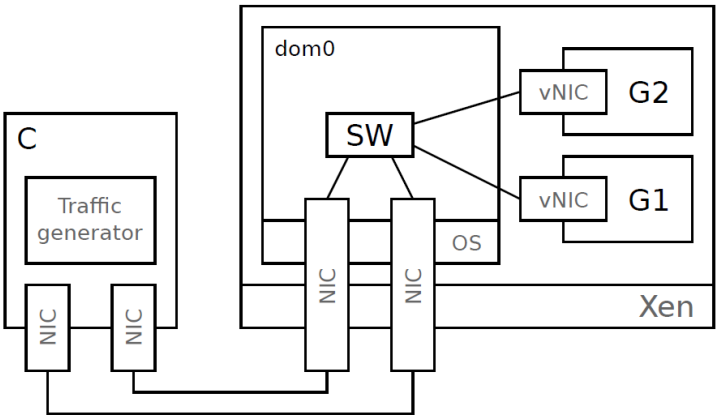

Figure 3.1 - Setup for Xen network performance measurements

Details of the system:

- Hypervisor: Xen 4.4.0
- Host (i.e. dom0) and guest OS: Ubuntu 14.04 LTS
- dom0 vCPUs: 4 (pinned to physical CPU cores)
- Guest vCPUs: 2 for each guest
- Guest memory: 2 GiB

3.1.4 Setup for QEMU-KVM measurements

Similar to the setup for Xen experiments, the measurements are performed using two VMs. However, an important difference is that there is no virtualized control domain, since the hypervisor runs on top of a host OS (except for the KVM kernel module). Thus, the bridge/switch component (SW) is also operated within the host OS. The setup is illustrated in Figure 3.2.

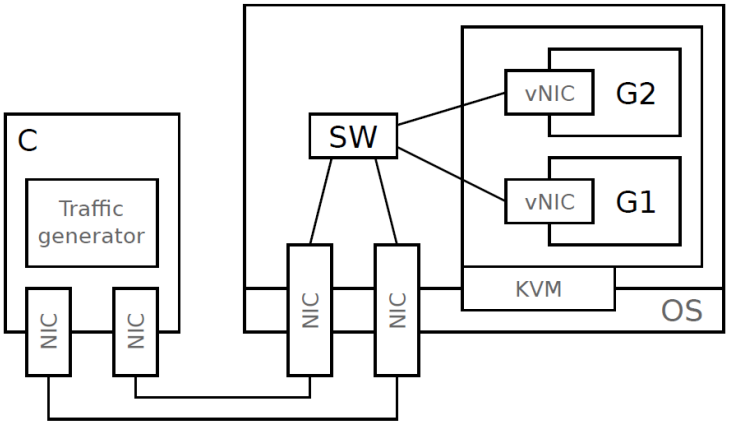

Figure 3.2 - Setup for KVM network performance measurements

Details of the system:

- Hypervisor: QEMU 1.6.2 when the VM uses DPDK, and QEMU 2.0.0 in the other cases
- Host and guest OS: Ubuntu 14.04 LTS
- Guest vCPUs: 2 for each guest
- Guest memory: 2 GiB

3.2 Experiments and Results

The Pktgen-DPDK tool (version 2.7.4) was used to generate a large amount of small (64 byte) packets, as we are interested in network performance in terms of pps (packets-per-second).

The traffic generator and the receiver is the same process, running on the physical machine denoted with C in Figure 3.1 and Figure 3.2. Traffic in and out of C flows on separate 10-Gigabit links, with the switch (SW) handling the forwarding using OpenFlow rules. The switch configuration is quite simple: only 4 rules are needed to forward all directions correctly. Actually in the measurements an even simpler set was used since the traffic was uni-directional. 2 rules were used with match fields on in_port and dst_ip.

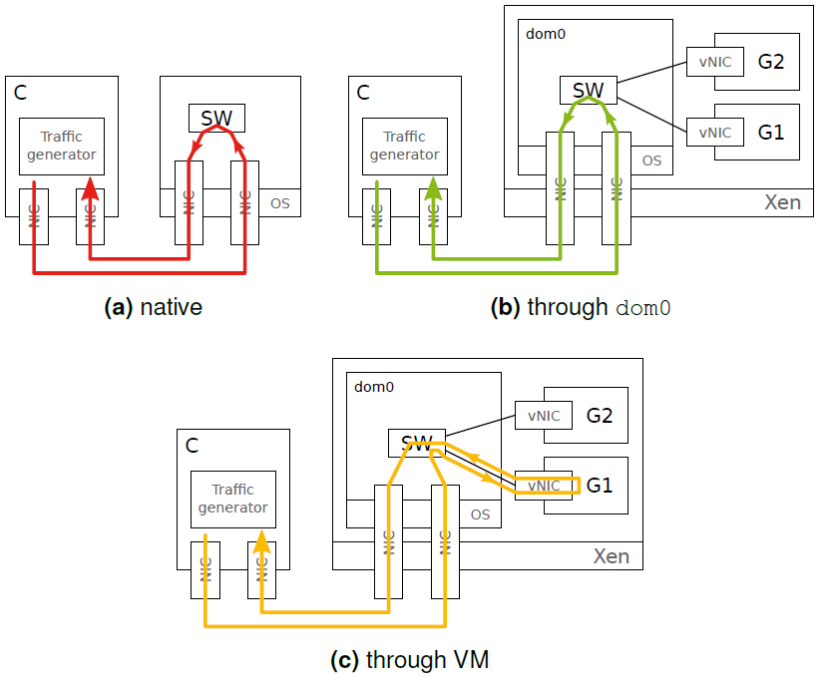

Figure 3.3 - UDP measurement scenarios

For Xen, three different scenarios are investigated, as can be seen in Figure 3.3. First, as a baseline, we measure the performance of a native system without virtualization. The packets are generated on C and then routed to the other system, where the SW redirects them back to C on a different link. For the second scenario the path for the traffic is

the same, except that SW now runs in a virtualized context (i.e. on dom0). In the third scenario the traffic also enters a guest VM before being turned back (using DPDK testpmd sample application). Unfortunately, OVDK does not support the bridging of Xen virtual interfaces, thus the third scenario is only evaluated using OVS. For KVM, the scenarios are analogous, except that there are no dom0 measurements.

3.2.1 Xen Results

The results of the tests on Xen can be seen in Figure 3.4. All measurements are performed four times – the average of the measured values are displayed, along with an error bar indicating the standard deviation.

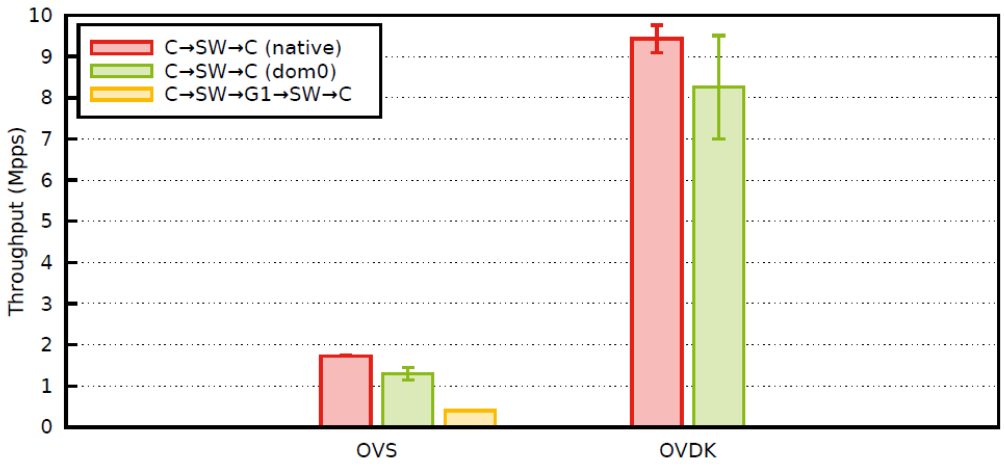

Figure 3.4 - Xen UDP 64-byte results

As expected, with the regular OVS (i.e. without DPDK on dom0) the results show poor performance, even without the overhead of virtualization. The use of OVDK brings a 5-6 x increase in performance.

The overhead of virtualizing dom0 is clearly visible, when comparing the results for the first (native) and second (dom0) scenario: performance drops to 75% for OVS and 88% for OVDK. Adding a VM to the path further impairs the performance in the case of OVS.

Later measurements with the Netmap/VALE [11] I/O acceleration framework and soft switch showed that Xen can reach approximately 5 Mpps in the C->SW->G1->SW->C path. This result can be compared to the OVDK/KVM-IVSHMEM results since Netmap uses a similar approach for the communication between the switch and the guest environment. That is still somewhat lower than the OVDK/KVM results, but it shows that, also for Xen, the accelerated environment greatly increases the available bandwidth (~ 15x increase).

3.2.2 KVM Results

As opposed to the case with Xen, OVDK is able to handle QEMU-KVM guests. The OVDK documentation offers two sample configurations [12] to achieve this. One uses QEMU IVSHMEM to communicate with guests via a single port, the other one uses a DPDK-accelerated vHost mechanism (running in user space) with two ports (one for each direction). In order to be consistent with our other measurements, our OVDK configurations include a third case, where user-space vHost is used with a single port. The results for the various scenarios and configurations are summarized in Figure 3.5 where "Userspace vHost x2" refers to the vHost configuration with 2 ports.

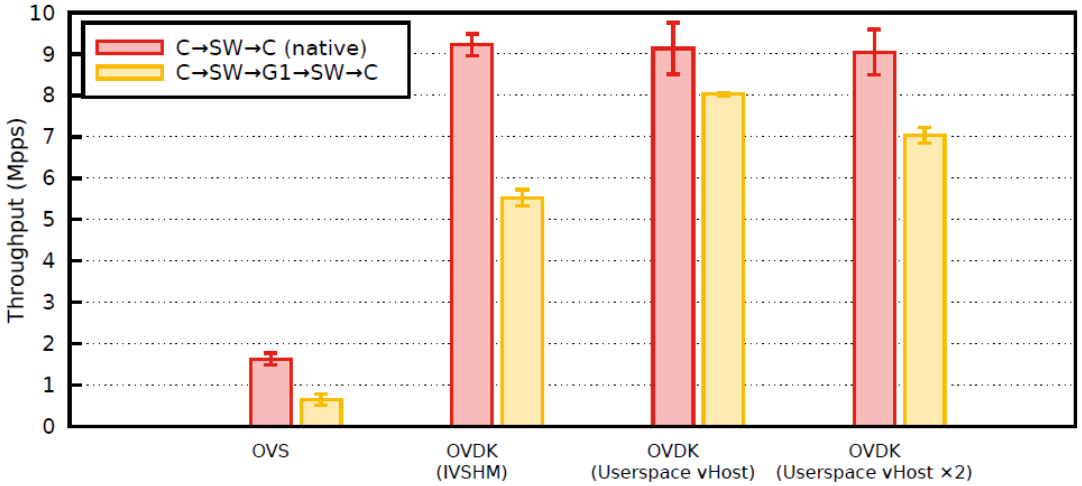

Figure 3.5 - KVM UDP 64-byte Results

Again, OVDK achieves better performance – around 5.5 x the performance of OVS in the native scenario, and an 8-12 times improvement for the various configurations through the VM. Not surprisingly, the three OVDK configurations yield almost identical results when the VM is not involved in traffic forwarding. Otherwise, the single-port user-space vHost solution performs the best. Here, the overhead of the VM is the most tolerable: it achieves almost 90% of the native throughput. The use of separate (vHost-backed) ports for ingress and egress fails to improve. Interestingly, when the second port is not used (i.e. QEMU emulates two guest ports, but the VM is configured to use only one for both ingress and egress), the performance is identical to the case where it uses both.

3.3 Conclusions

Although this testing was limited in comparing Xen and KVM with DPDK acceleration, a few results are important:

- Since Xen uses a dom0 hypervisor it is slower in executing the virtual switch and this way all virtual switch based VNFs. This is valid for both DPDK and generic case, although the difference is smaller when DPDK is used in dom0.

- Most probably due to this overhead Xen is also slower in VNF I/O towards the virtual machines, although here only the generic case was initially available for comparison. Later measurements confirmed that there is indeed a difference between Xen and KVM, but also that the difference is smaller in the accelerated setup.
- It is also visible that the I/O overhead can be significantly reduced with DPDK (and likely with other frameworks using similar techniques), down to around 10%, which means the overhead of the introduction of a Virtual Machine in the data path (NIC to host to guest and return) can largely be contained.

4 Initial UN Benchmarking

This section introduces the first tests where actual functions execute on top of the basic building blocks that were in focus in the previous sections. One major difference is that the traffic must now also flow between the vSwitch and the VNFs.

4.1 Prototype VNFs

The prototype VNFs used for this testing where implemented based on the Intel® Data Plane Performance Demonstrators (DPPD) [13]. Intel DPPD is released under a 3-clause BSD license. It provides a framework to implement packet processing tasks using DPDK and run them in various conditions and configurations (number of cores, multiple tasks on a same core) for performance impact evaluation.

Support for ports backed by DPDK software rings was added to DPPD so that it can run the prototype VNFs connected to the Universal Node VSE (vSwitch) using software queues residing in shared memory in addition to driving physical or virtual (Virtio) NIC ports. This uses the IVSHMEM [14] mechanism found in QEMU to share host memory with VMs. In theory, it allows a zero-copy transfer of packets between the VSE and the packet processing application running in the VM. A configuration using user-space vHost Virtio is also possible but hasn't been tested at this stage.

The following prototype VNFs were used for the tests covered in sub-sections 4.3, 4.4 and 4.5:

- "handle_none": This is the simplest DPPD workload configuration. It simply loops all packets received on one port back to the same port or to another port, not actually modifying or acting upon the packets. This will remain a valid performance baseline to distinguish the impact of the UN platform from the specific implementation of a VNF.
- "lwAFTR": A lwAFTR datapath component implemented in DPPD (lw_aftr.cfg configuration). This is a component of the Lighweight 4over6 architecture.
- "BNG": A DPPD configuration that implements a prototype of Broadband Network Gateway.

4.2 Test Setup

4.2.1 UN Hardware Platform

Tests were performed on the same dual socket Intel® Xeon® based systems with processors of the "Ivy Bridge" family as were used for the physical-to-physical tests in section 2.1.2. Here are relevant details:

- Intel® Xeon® E5-2697 v2 @ 2.7 GHz
- 10Gb "Niantic" (82599) network interfaces
- CPU frequency was set at constant 2.7 GHz (Turbo, P-States and C-states disabled)

4.2.2 UN Software

4.2.2.1 rofl-core

The ***R**evised **O**penF**l**ow **L**ibrary* (ROFL) [15] is a dependency of xDPd and provides OpenFlow support. The following commit and configure command line were used to build the library:

```
commit 0e8fe5e305c64cf1cbcd92b1f0c5300dda6d5f52 from Feb 2 2015
../configure –with-pipeline-platform-funcs-inlined –with-pipeline-lockless
```

4.2.2.2 xDPd

The following commit and configure command line were used to build xDPd, using DPDK 1.7.1:

```
commit 1dd9f6cb917f6cafa01564491506520f461f8758 from Feb 16 2015
../configure –with-hw-support=gnu-linux-dpdk –disable-silent-rules
```

4.2.2.3 DPPD

DPPD release v015 was used, built with DPDK 1.7.1.

4.2.2.4 QEMU

In order to use IVSHMEM to share DPDK rings between the host and the VMs, the modified QEMU version from the Intel® DPDK vSwitch project was used. It can be found under the qemu directory of the DPDK vSwitch 1.1 release archive [16].

4.2.3 Traffic Generation

The tests performed do not require specific testing equipment. All test traffic can be generated using a similar hardware platform as for the UN, and using the Pktgen software from https://github.com/Pktgen/Pktgen-DPDK/.

4.3 "handle_none" VNF

In this test case, traffic consisting of 64 bytes Ethernet packets (smallest packet size) is sent to two physical ports of the UN and is then steered towards the "handle_none" VNF that performs a simple bi-directional loopback. This represents the simplest possible middle-box function and therefore provides a good baseline for the performance of the platform and the VSE. It should however be noted that, since it doesn't access packet data, its cache usage pattern may significantly differ from that of more useful functions.

From the outside of the UN, this is the exact same setup as used in earlier tests where xDPd was forwarding traffic between two physical ports (see section 2.1.2), but internally, traffic now goes through the "handle_none" VNF, running as a VM. The VM is provided with 2 dedicated CPU cores (QEMU threads pinned to specific cores). The xDPd soft switch was configured with 3 packet processing CPU cores (CORE_MASK=0x0f, one core being assigned to the control plane handling). Very simple flow rules, only based on the port packets are received on, were used to steer traffic through the VM, over software rings in memory shared between host and guest (IVSHMEM).

The performance measured for various packet sizes is shown on Figure 4.6.

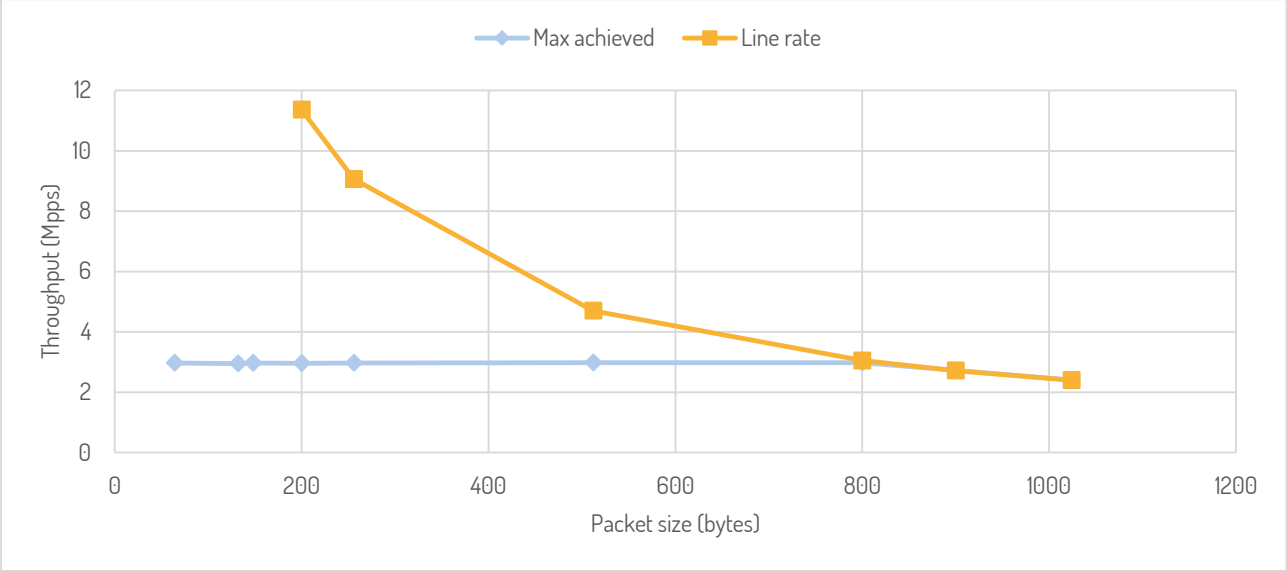

Figure 4.6 – Throughput of xDPd with "handle_none" VNF

The performance degradation when introducing the VNF on the data path is significant, going from a peak throughput of 12.7 Mpps (Figure 2.5) to just 3 Mpps.

Assigning a 4th processing core to the soft switch did not improve the throughput, which is surprising given that we now have 4 ports (2 physical and 2 virtuals) that the vSwitch has to handle and we could verify that it does indeed distribute processing of the 4 ports to the 4 provided cores on a 1-to-1 basis. Note that the VNF implementation itself is also not the bottleneck as it was verified to run at line rate when directly accessing the NICs, without vSwitch in between.

Of course, introducing the VNF also implies that the vSwitch performs an additional complete switching step when the traffic comes back from the VNF, but this does not justify such performance degradation especially when dedicating one CPU core to each vSwitch port. Also, the results measured with OVDK and KVM (see section 3.2.2) show that a higher throughput is definitely achievable for this case[3]. Further bottleneck analysis will need to determine the cause of this significant performance degradation. Impacting factors might be, e.g., remaining locks when accessing shared data, cache trashing, interference of bulk processing of the different ports in the soft switch and with the one happening in the VNF.

[3] Measurements of this exact setup with OVDK were not conducted as OVDK is not actively maintained anymore. Future measurements will however cover its successor, OVS with DPDK netdev implementation.

It should also be noted that, contrary to the case of section 2.1.2 tests (physical to physical), in this case, when the offered load keeps increasing beyond the capacity of the system, the throughput actually decreases a little before bottoming out to its final value. This can be seen on Figure 4.7 which shows the measured throughput as a function of the offered load for two selected packet sizes (64 and 512).

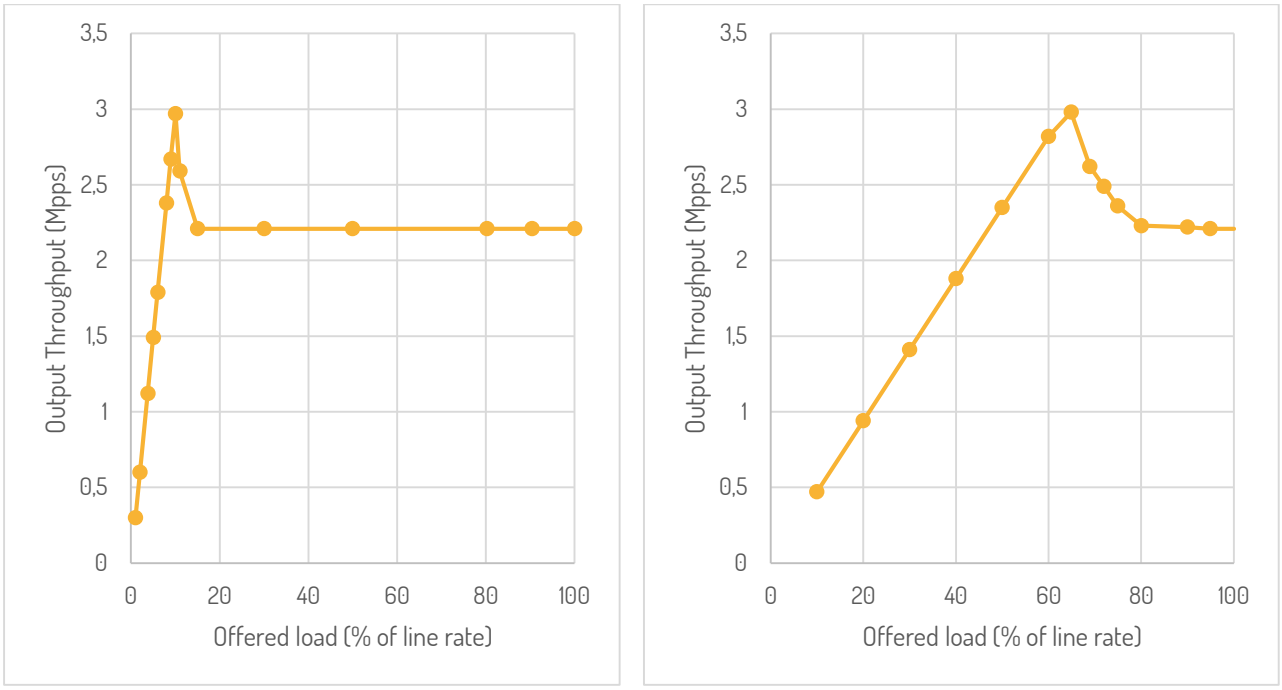

Figure 4.7 - xDPd with "handle_none" VNF (left: 64 byte packets , right: 512 byte packets)

This is likely due to a sort of accumulation effect of the software queues between the vSwitch and the VM, which introduces additional overhead when the vSwitch eventually needs to drop packets in addition to processing others, until it has no more processing cycles to read from the NIC queue, at which point the packets start to be dropped by the NIC itself, without any further overhead for the vSwitch, thereby reaching the plateau that is observed. This will need to be verified in future detailed analysis.

4.4 lwAFTR VNF

The lwAFTR VNF implements the lwAFTR component of the Lightweight 4over6 architecture [17] and was introduced in D5.3 [3]. In this test case, the VNF handles 2 directions of traffic and performs IPv6 encapsulation on one direction, and decapsulation in the other. For both directions, a lookup in the binding table is also performed.

The test traffic includes randomly selected entries from the binding table, and a random port from the port set associated with the selected entry. The traffic consists of only packets of the smallest size given the IPv6 encapsulation: Considering the sizes of the Ethernet header (14 bytes), the IPv6 header (40 bytes), the IPv4 header (20 bytes) and an empty UDP payload (8 bytes), the smallest tunnel packet consists of 82 bytes, to which the

Ethernet overhead must be added (7 bytes preamble, 1 byte SFD, 4 bytes FCS and an inter-packet gap of 12 bytes). On a 10GbE link, this gives a maximum packet rate of 11,792 Mpps, which we hence consider as the "line rate" for this test.

Note: Ethernet packets are at least 64 bytes long (including the FCS). On the IPv4 side where packets are smaller, padding may be introduced. When encapsulating such padded packet, care must be taken to remove the padding before applying the IPv6 encapsulation, otherwise the theoretical line rate could never be achieved: the smallest packet would be 64 + 40 bytes yielding a maximum rate of 10,08 Mpps.

For the lwAFTR tests, traffic was generated using DPDK-PktGen playing pre-generated PCAP trace files. A script was created (see Annex 2) to generate PCAP files with random traffic according to a provided binding table. The input file containing the binding table should be the same as used by the lwAFTR at run-time, and can be generated by the ipv6_tun_bindings.pl script provided in the helper-scripts/ipv6_tun folder of the DPPD archive). To generate PCAP files for both directions, the following commands can be used:

```
./gen_4over6.pl -inet -in=ipv6_tun_bind.csv -out=lwaftr_inet.pcap
./gen_4over6.pl -tun -in=ipv6_tun_bind.csv -out=lwaftr_tun.pcap
```

The generated PCAP files can then be specified to DPDK-PktGen on a per port basis.

4.4.1 lwAFTR VNF "Bare-metal"

This is a reference test of the lwAFTR VNF running without any hypervisor and directly accessing the NIC. This means that the NIC is dedicated to the VNF. This model is not aligned with the requirements and architecture for the UN but can be a useful reference.

DPPD was configured with one direction of the tunnel handled by one core. A complete lwAFTR instance therefore uses 2 cores. The two cores were located on two physically different cores (Hyperthreading not used).

Table 2 shows the throughput achieved in each direction as a function of the size of the binding table.

Binding table entries:	**1k**	**5k**	**10k**	**25k**	**50k**	**100k**	**200k**	**500k**	**1M**
Encap (Mpps) (% line rate)	11.79 (100%)	11.79 (100%)	10.45 (88%)	11.25 (95%)	11.79 (100%)	11.79 (100%)	11.75 (99.6%)	11.03 (95.5%)	10.80 (91.6%)
Decap (Mpps) (% line rate)	11.79 (100%)	11.79 (100%)	11.79 (100%)	11.79 (100%)	11.79 (100%)	11.79 (100%)	11.79 (100%)	11.79 (100%)	11.79 (100%)

Table 2 - lwAFTR Bare Metal Performance in function of the binding table size

Aside from an anomaly with small binding tables (10k to 25k entries) that can be circumvented by over-dimensioning the table, the results show that a pair of cores can handle line-rate traffic of a 10Gb link (both directions) for binding tables of up to 200k entries.

4.4.2 lwAFTR VNF Running in a VM

This test consists in running the same lwAFTR function in a VM with traffic steered to the VM through the xDPd VSE. The flow rules were very simple, only based on the port the packet is received on. The VM is given two CPU cores and the QEMU threads are pinned to their own physical CPU core.

The bi-directional throughput achieved in an otherwise similar setup as for the bare-metal test was 2.68 Mpps (1.34 Mpps in each direction).

This is slightly lower than the 3 Mpps measured for the "handle_none" VNF with similar packet size (we have 64 bytes in one direction and 82 in the other). The VNF itself is not the limiting factor because we dedicate 1 CPU core to the handling of each direction of the traffic and the bare-metal test showed that the VNF implementation can handle a higher throughput. Further bottleneck analysis will need to be performed once the handle_none case is fully understood.

4.5 BNG

The intention of the broadband network gateway (BNG) use case is to study a more complex function combining several elementary ones that exercise various platform aspects at the same time (packet processing and modification, lookups into large tables, etc.). More complex processing pipelines like that of a BNG also require load distribution mechanisms to support them, which is a topic of interest for the UN work package.

Earlier results and analysis about an existing DPDK based BNG implementation can be found in [18] and [19]. This implementation will serve as basis for future BNG related work in the project. Those earlier results however relate to running the BNG bare metal and after careful fine-tuning. At time of writing, no results are available yet for the BNG use case on the UN environment. An update of the bare-metal results based on a more recent version of the BNG implementation is however provided below.

The fine-tuned results can be seen as a target to try to approach when doing automated deployment of similar network functions.

The BNG implementation is provided by a specific DPPD [13] (see section 4.1) configuration that provides the functionality depicted on Figure 4.8.

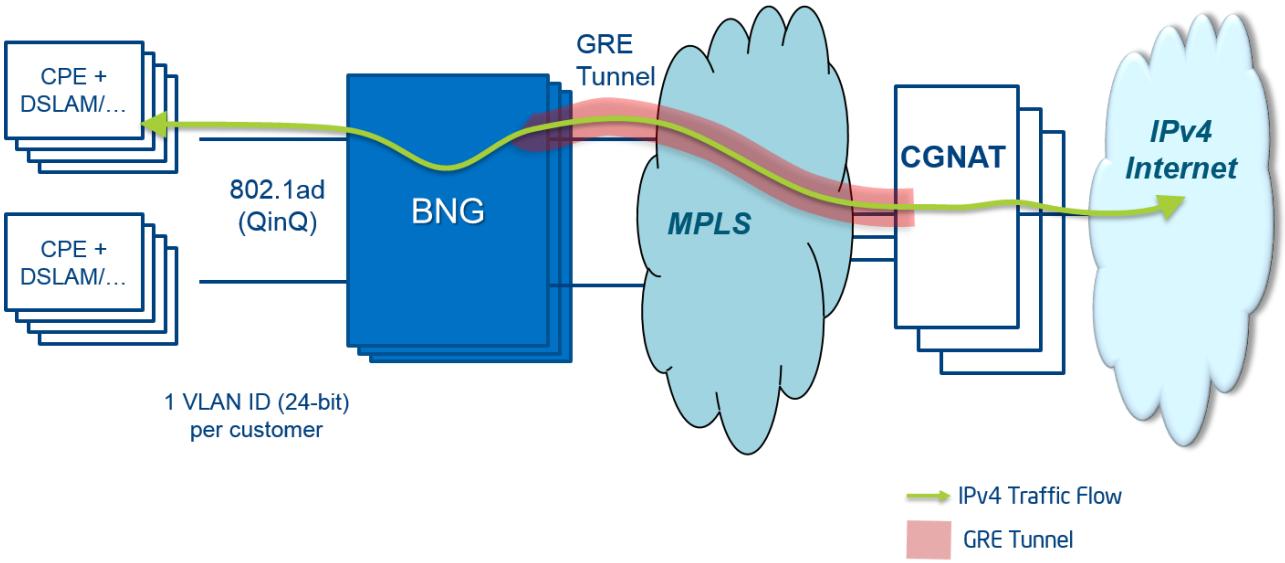

Figure 4.8 – DPPD Broadband Network Gateway (BNG) functionality

This BNG implementation provides mapping between the customer traffic identified by the QinQ VLAN ID and the corresponding GRE tunnel key. For traffic originating from the CPEs, a longest prefix match is performed on the target IP address to identify the appropriate CGNAT, as well as the MPLS tag to use.

Two flavours of the BNG application have been tested. The first one, introduced in [18] and further refined in [19], does not include any QoS functionality. A second flavour adds a QoS element that limits the rate at which users can transmit and receive traffic. The QoS element introduces traffic policing that controls the influence between flows of different users, and also helps to smooth out the oversubscription or jitter present in the incoming stream.

In both cases tested, the network ports are 10GbE ports on Intel® 82599 controllers. The setup uses four 10GbE ports on two network interface cards: 2 ports towards CPEs, 2 ports towards the core network (MPLS side towards CGNATs).

4.5.1 Packet Throughput

Because additional optimizations were implemented in DPPD since the publication of the reports [18] and [19], an update of the results is provided here, based on DPPD v015, the same version that is used in the other tests in this section, and running on the same processor (Intel® Xeon® E5-2697 v2 @ 2.7GHz).

Figure 4.9 shows the total throughput achieved for varying packet sizes (upstream and downstream sizes being different, they are both indicated), for the two flavours of the BNG, without and with QoS. The Theoretical Max values correspond to line rate on 4 interfaces with the specified packet sizes, taken at the core network side (including MPLS and GRE), indicated as "downstream" on the figure. This is indeed the side that has the largest overhead and hence determines the highest achievable packet rate.

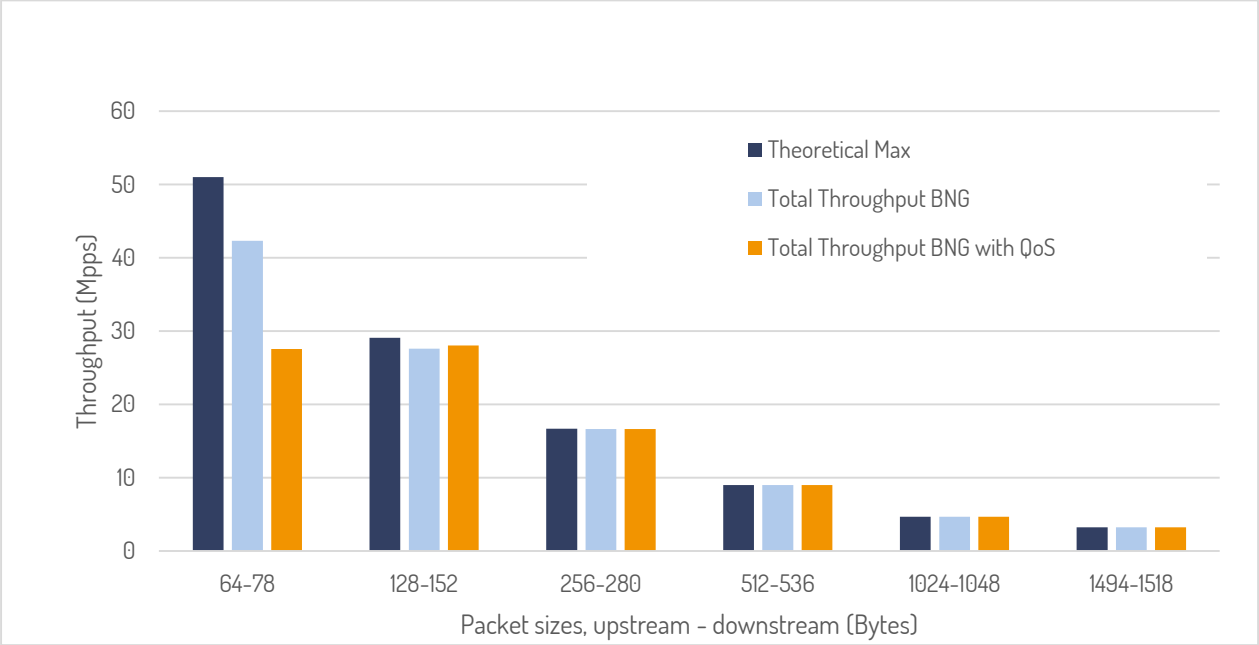

Figure 4.9 - BNG Throughput without and with QoS

As shown in Figure 4.9, the BNG implementations essentially achieve line rate for packets of size 128 bytes and above. For the smallest size packets, the throughput reaches 83% of line rate for the flavour without QoS while the one with QoS only reaches 54%.

The function consists of several smaller tasks that execute on a number of CPU cores. A total of 12 CPU cores were used for the flavour without QoS, whereas 22 cores were used to execute the QoS flavour.

4.5.2 Additional Results

In addition to the description of the BNG use case, the implemented architecture and the throughput measurements, the reports [18] and [19] also explored the performance impact of various parameters on simpler configurations. This input was used to drive the architecture of the BNG and should provide useful input in identifying the parameters to take into account when doing an automated deployment of VNFs by the UN Local Orchestrator. A summary some of these conclusions is given in Annex 1.

4.5.3 Next Steps

The next steps with the BNG workload will be first to run the implementation on the UN, as a real VNF. Currently, the BNG is run as a monolithic single process application (although internally, it consists of multiple finer grained tasks), and a subsequent step will consist in splitting the BNG into several smaller functions that will be executed as individual VNFs chained together. This will allow to compare the performance obtained when this chain is somehow pre-configured by directly connecting the VNFs together to implement the BNG, with the more flexible case where traffic goes back to the vSwitch between each VNF to see how much performance this flexibility costs. This case will support an investigation on how this VNF chaining could be optimized on the UN. Finally, this will also support an investigation on how to move some of the finer grained VNFs into the vSwitch.

5 Conclusions

Initial measurements were mostly done on the xDPd soft switch and indicate that the physical-to-physical performance is within the targets set in D5.1 but also that some bottleneck exists and needs to be identified for the case where traffic is steered through VMs.

However, the main goal of xDPd was to be highly portable and, as a result, it follows an architecture that may not accommodate all sorts of optimizations. Consequently, in parallel with further analysis and improvements on the xDPd software switch, we will also start doing detailed measurements with other switch implementations like Open vSwitch. This will serve as comparison ground but OVS could also offer an alternative VSE implementation, for UN benchmarking.

xDPd, on the other hand, being portable across a wide range and scale of devices, is well positioned for VSE specific implementations like supporting VNF within the VSE, or experimenting with OpenFlow extensions for load-balancing as proposed in D5.2. Depending on their relative performance and suitability for implementing additional functionality within the UNIFY project, one or the other VSE alternative could be used.

The evaluation of a Run-Time Code Generation optimization for OpenFlow rules and actions showed that this is a promising technique, immediately valuable for the cases where performance on the smallest packet sizes matters, and otherwise freeing resources for other processing on the node.

Further investigation into the existing bottleneck(s) in the xDPd-to-VM data transport is going on. At the time of writing, there are indications that it could be due to a mistake in batching frames upwards from xDPd. As other tests with OVS indicated the bottleneck of going through a virtual machine should be fairly small, around 10% in terms of transaction speed.

The ongoing work targeting the split of the BNG function into finer grained VNFs will likely also split out dataplane from control in a way that the data plane transmission can still take place within the accelerated zone, while the VMs are function-specific controllers holding the session state.

Further options to speed up the pure forwarding is the shift of the VSE into a switching ASIC or network processor (NPU) that is either located on the board or within a NIC.

List of abbreviations and acronyms

Abbreviation	Meaning
API	Application Programming Interface
BNG	Broadband Network Gateway
CGNAT	Carrier Grade Network Address Translation
CP	Control Plane
CPE	Customer Premise Equipment
DP	Data Plane
DTLB	Data Translation Lookaside Buffer
DUT	Device Under Test
IOMMU	I/O Memory Management Unit
IOTLB	I/O Translation Lookaside Buffer
IVSHMEM	QEMU mechanism for sharing host memory with VMs
LSI	Logical Switch Instance
LXC	LinuX Containers
NF	Network Function
NF-FG	Network Function Forwarding graph
NIC	Network Interface Card (often refers to a physical network port regardless of its presence on a card)
OF	OpenFlow
OVS	Open vSwitch
UN	Universal Node
URM	Unified Resource Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNF EE	VNF Execution Environment
VSE	Virtual Switching Engine
VLAN	Virtual LAN
VXLAN	Virtual Extensible LAN

References

[1] UNIFY Consortium, "UNIFY Deliverable 5.1," 2014.

[2] UNIFY Consortium, "UNIFY Deliverable 5.2," 2014.

[3] UNIFY Consortium, "UNIFY Deliverable 5.3," 2014.

[4] BISDN, "The eXtensible DataPath Daemon (xDPd)," [Online]. Available: http://www.xdpd.org/. [Accessed 23 May 2014].

[5] [Online]. Available: http://www.xenproject.org/. [Accessed 30 April 2014].

[6] [Online]. Available: http://www.qemu.org/. [Accessed 30 April 2014].

[7] "Ryu SDN Framework," [Online]. Available: http://osrg.github.io/ryu/.

[8] S. NILSSON and G. KARLSSON, "Fast address lookup for internet routers," *Broadband Communications,* pp. 11-22, 1998.

[9] "Open vSwitch," [Online]. Available: http://openvswitch.org/. [Accessed 22 April 2014].

[10] "Intel® DPDK vSwitch," [Online]. Available: https://github.com/01org/dpdk-ovs. [Accessed 22 April 2014].

[11] L. Rizzo, "netmap - the fast packet I/O framework," [Online]. Available: http://info.iet.unipi.it/~luigi/netmap/. [Accessed 26 June 2014].

[12] „Intel DPDK vSwitch Documentation – Sample Configurations," [Online]. Available: https://github.com/01org/dpdk-ovs/tree/development/docs/04_Sample_Configurations.

[13] „Intel Data Plane Performance Demonstrators (DPPD)," Intel, [Online]. Available: https://01.org/intel-data-plane-performance-demonstrators. [Zugriff am 8 10 2014].

[14] C. Macdonell, "Inter-VM shared memory PCI Device," [Online]. Available: http://thread.gmane.org/gmane.comp.emulators.kvm.devel/49528. [Accessed 2 3 2015].

[15] BISDN GmbH, „The Revised OpenFlow Library," [Online]. Available: http://roflibs.org. [Zugriff am 16 Feb 2015].

[16] Intel Corporation, "Intel® DPDK vSwitch R1.1," [Online]. Available: https://01.org/sites/default/files/downloads/packet-processing/openvswitchdpdk.1.1.1.0-27.gz. [Accessed 9

March 2014].

[17] Y. Cui, Q. Sun, M. Boucadair, T. Tsou, Y. Lee and I. Farrer, "Lightweight 4over6: An Extension to the DS-Lite Architecture," 6 June 2014. [Online]. Available: https://tools.ietf.org/html/draft-ietf-softwire-lw4over6-10. [Accessed 4 November 2014].

[18] Intel Corporation, "Network Function Virtualization: Virtualized BRAS with Linux* and Intel® Architecture," 2014. [Online]. Available: https://networkbuilders.intel.com/docs/Network_Builders_RA_vBRAS_Final.pdf.

[19] Intel Corporation, „Network Function Virtualization: Quality of Service in BRAS with Linux* and Intel® Architecture®," [Online]. Available: https://networkbuilders.intel.com/docs/Network_Builders_RA_NFV_QoS_Aug2014.pdf.

[20] Open Networking Foundation, "OpenFlow Switch Specification v1.3.3," 13 September 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.3.pdf. [Accessed 10 February 2014].

Annex 1 - Conclusions from BNG Fine Tuning Reports

This annex summarizes some of the conclusions from reports [18] and [19]:

- Impact of Virtualization - The study shows that, starting with the second generation of Intel Xeon Processor E5 ("Ivy Bridge"), there is essentially no cost associated with running the BNG prototype in a virtual machine compared to running it on the host system. In earlier processor generations, the IOTLB did not natively support huge pages and this was causing a performance degradation when the IOMMU had to translate many virtual I/O addresses to host physical addresses. This specific aspect only applies when the VM directly accesses the NIC. However, the results also show that there was no other impact of virtualization for the considered test cases.
- Placement of tasks on sockets – The first report provides an exhaustive study of the different configurations of load balancers, receiving packets from the network interfaces and feeding worker threads that eventually transmit the packets out over another network interface. The study concludes that, for such a case, it is better to have the receiving tasks executing on cores of the same socket where the network interface is directly connected, even if the interface out of which the packets must be transmitted is on the other socket.
- Performance Stability and Packet Buffer Order – The reports show that the fact of shuffling packet buffers in their pool has a performance impact, and that, as a result, an application will see its performance degrade over time if it's mostly keeping the packet buffers in order but sometimes introduces disruption to the sequence due to relatively exceptional packet handling. In a dynamic system like the UN where multiple NF-FGs can be deployed and different flows steered through the different graphs and functions, it is very unlikely that the order of packet buffers can be preserved. This should be accounted for in future work to avoid measuring an unrealistic situation.
- Impact of DTLB misses (Huge Pages) - The reports also demonstrates the impact of DTLB misses and the fact that hyperthreaded logical cores share the same TLB and hence are subject to performance degradation with smaller working sets. This stresses the importance for all but the simplest workloads to make use of huge memory pages in order to reduce the pressure on the TLB. Note however that this is the normal setup for DPDK based applications.

Annex 2 - lwAFTR test traffic generation script (gen_4over6.pl)

```
#!/usr/bin/perl

use strict vars;
use Getopt::Long;
use Pod::Usage;
use Net::Pcap;
use Text::CSV;
use Net::Frame::Layer;
use Net::Frame::Layer::ETH qw(:consts);
use Net::Frame::Layer::IPv6 qw(:consts);
use Net::Frame::Layer::IPv4 qw(:consts);
use Net::Frame::Layer::UDP;
use Socket qw(AF_INET AF_INET6 inet_ntop inet_pton);

use constant NUM_PACKETS => 30000;

use constant ETHER_ADDR_LEN => 6;
use constant ETHER_TYPE_LEN => 2;
use constant ETHER_HDR_LEN => ( 2 * ETHER_ADDR_LEN ) + ETHER_TYPE_LEN;
use constant ETHER_STATIC_MAC => "78acddddddd";

use constant UDP_HDR_LEN => 8;
use constant UDP_STATIC_PORT => 0x6666;

use constant IPv6_HOP_LIMIT => 4;
use constant IPv6_STATIC_IP => "2222:2222:2222:2222:2222:2222:2222:2222";

use constant IPv4_TIME_TO_LIVE => 32;
use constant IPv4_STATIC_IP => "68.68.68.68";

srand;

my $type = 'tun';
my $pkt_count = NUM_PACKETS;
```

```
GetOptions(
        'inet' => sub { $type = 'inet'},
        'tun' => sub { $type = 'tun'},
        'count=i'       => \$pkt_count,
        'in=s' => \(my $in = 'ipv6_tun_bind.csv'),
        'out=s' => \(my $out = 'output.pcap'),
        'size=s' => \(my $size = 0)
) or exit;

my $pcap = pcap_open_dead( DLT_EN10MB, 65535 );
my $dumper = pcap_dump_open($pcap, $out ) or die 'Could not create output file: ' . $out;

if( $type eq 'inet' ) {
        gen_inet_pcap( $in, $pkt_count );
}
if( $type eq 'tun' ) {
        gen_tun_pcap( $in, $pkt_count );
}

pcap_close( $pcap );

# Trim string
sub trim {
        my ( $str ) = @_;

        $str =~ s/^\s+|\s+$//g;

        return $str;
}

# Generate random port based on $port and $port_mask
sub rand_port {
        my ( $port, $port_mask ) = @_;

        return ( $port | int( rand( 0xFFFF ) & $port_mask ) );
```

```
}

# Generate packet originating from CPE
sub gen_tun_packet {
        my ( $sz, $ether, $ipv6, $ipv4, $udp ) = @_;

        my $hdr_ether = Net::Frame::Layer::ETH->new(
                src => $ether->{'src'},
                dst => $ether->{'dst'},
                type => NF_ETH_TYPE_IPv6
        )->pack;

        my $hdr_ipv6 = Net::Frame::Layer::IPv6->new(
                nextHeader => NF_IPv6_PROTOCOL_IPIP,
                hopLimit => IPv6_HOP_LIMIT,
                src => $ipv6->{'src'},
                dst => $ipv6->{'dst'},
                payloadLength => $sz + NF_IPv4_HDR_LEN + UDP_HDR_LEN
        )->pack;

        my $hdr_ipv4 = Net::Frame::Layer::IPv4->new(
                length => $sz + UDP_HDR_LEN + NF_IPv4_HDR_LEN,
                ttl => IPv4_TIME_TO_LIVE,
                protocol => NF_IPv4_PROTOCOL_UDP,
                src => $ipv4->{'src'},
                dst => $ipv4->{'dst'}
        )->pack;

        my $hdr_udp = Net::Frame::Layer::UDP->new(
                src => $udp->{'src'},
                dst => $udp->{'dst'},
                length => $sz + UDP_HDR_LEN
        )->pack;

        my $pkt = pack( "H*", "de" x $sz );
        $pkt = $hdr_ether . $hdr_ipv6 . $hdr_ipv4 . $hdr_udp . $pkt;
```

```
        my $pkt_size = length( $pkt );

        my $hdr = {
                tv_sec => 0,
                tv_usec => 0,
                len => $pkt_size,
                caplen => $pkt_size
        };

        return ( $hdr, $pkt );
}

# Generate packet originating from the internet
sub gen_inet_packet {
        my ( $sz, $ether, $ipv4, $udp ) = @_;

        my $hdr_ether = Net::Frame::Layer::ETH->new(
                src => $ether->{'src'},
                dst => $ether->{'dst'},
                type => NF_ETH_TYPE_IPv4
        )->pack;

        my $hdr_ipv4 = Net::Frame::Layer::IPv4->new(
                length => $sz + UDP_HDR_LEN + NF_IPv4_HDR_LEN,
                ttl => IPv4_TIME_TO_LIVE,
                protocol => NF_IPv4_PROTOCOL_UDP,
                src => $ipv4->{'src'},
                dst => $ipv4->{'dst'}
        )->pack;

        my $hdr_udp = Net::Frame::Layer::UDP->new(
                src => $udp->{'src'},
                dst => $udp->{'dst'},
                length => $sz + UDP_HDR_LEN
        )->pack;
```

```
        my $pkt = pack( "H*", "de" x $sz );
        $pkt = $hdr_ether . $hdr_ipv4 . $hdr_udp . $pkt;

        my $pkt_size = length( $pkt );

        my $hdr = {
                tv_sec => 0,
                tv_usec => 0,
                len => $pkt_size,
                caplen => $pkt_size
        };

        return ( $hdr, $pkt );
}

# Read CSV binding file
sub read_csv {
        my ( $file ) = @_;

        print "Reading CSV file...\n";

        my @rows;
        my $csv = Text::CSV->new ( { binary => 1 } )
                or die "Cannot use CSV: ".Text::CSV->error_diag ();

        open my $fh, "<:encoding(utf8)", $file or die $file . ": $!";
        while ( my $row = $csv->getline( $fh ) ) {

                inet_pton( AF_INET6, trim( $row->[0] ) ) or next;
                inet_pton( AF_INET, trim( $row->[2] ) ) or next;
                int(trim($row->[3])) or next;

                push @rows, {
                        ipv6 => trim($row->[0]),
                        mac => trim($row->[1]),
```

```
                ipv4 => trim($row->[2]),
                port => trim($row->[3])
            }
        }
        $csv->eof or $csv->error_diag();
        close $fh;

        return @rows;
}

# Generate packets originating from CPE
sub gen_tun_pcap {
        my ( $binding_file, $pkt_count ) = @_;
        my @csv = read_csv($binding_file);
        my $idx = 0;
        my $row;
        my $public_port = 0;

        print "Generating $pkt_count Tunnel packets...\n";

        my $max = @csv;
        for( my $i=0; $i<$pkt_count; $i++ ) {

            $idx = rand $max;
            $row = @csv[$idx];
            $public_port = rand_port( $row->{port}, 0x3f );

            my ( $hdr, $pkt ) = gen_tun_packet(
                $size,
                { src => $row->{mac}, dst => ETHER_STATIC_MAC },
                { src => $row->{ipv6}, dst => IPv6_STATIC_IP },
                { src => $row->{ipv4}, dst => IPv4_STATIC_IP },
                { src => $public_port, dst => UDP_STATIC_PORT }
            );

            pcap_dump( $dumper, $hdr, $pkt );
```

```
        }
}

# Generate packets originating from the internet
sub gen_inet_pcap {
        my ( $binding_file, $pkt_count ) = @_;
        my @csv = read_csv($binding_file);
        my $idx = 0;
        my $row;
        my $public_port = 0;

        print "Generating $pkt_count Internet packets...\n";

        my $max = @csv;
        for( my $i=0; $i<$pkt_count; $i++ ) {

                $idx = rand $max;
                $row = @csv[$idx];
                $public_port = rand_port( $row->{port}, 0x3f );

                my ( $hdr, $pkt ) = gen_inet_packet(
                        $size,
                        { src => ETHER_STATIC_MAC, dst => $row->{mac} },
                        { src => IPv4_STATIC_IP, dst => $row->{ipv4} },
                        { src => UDP_STATIC_PORT, dst => $public_port }
                );

                pcap_dump( $dumper, $hdr, $pkt );
        }
}
```