whitestack

AMD-DPDK WhiteCloud Traffic Tests

About this Document

This document is a report on the DPDK traffic tests performed in Dell servers using AMD processors and Intel network adapters. These tests were performed in order to inspect the maximum amount of traffic that can be output, in `Gib/s` and in `pkt/s`, by virtual machines running inside a WhiteCloud deployment to an external network while using a DPDK dataplane.

Our tests consisted in hosting several Ubuntu virtual machines on a single server and having them generate as much outwards traffic as possible, using the `pktgen` tool, a DPDK packet generation tool. The virtual computing and networking components and configurations needed to perform these tests were provided by Docker containers running Openstack services; these were part of a larger WhiteCloud (WhiteStack's Openstack distribution) Yungay deployment.

The objective of this report is to disclose the characteristics of the environment used for the tests, explain the different configurations used for WhiteCloud and OVS in order to get the most amount of outgoing traffic possible, display and explore the obtained results and, finally, present a brief discussion about the scope and limitations of the test, including some possible next steps.

Lab Environment

Physical Environment

The physical environment that was set up to run our tests was composed of a single `Dell PowerEdge R7525` server, `EPYC-05`, connected to two `Dell EMC Networking OS10` Switches. A diagram explaining the lab environment an its connections can be seen below.

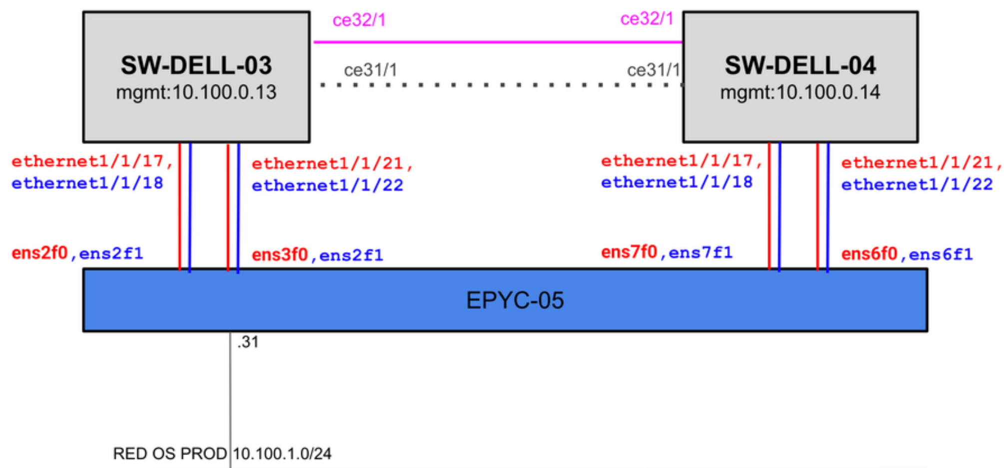


Figure 1 - Lab Physical Environment

Each `ens` interface pair shown connected to the server on *Figure 1*, red for `ensXf0` and blue for `ensXf1`, represent both ports of an Intel E810-C Network adapter with a `100Gb` combined data rate. Four network adapters were used on this server, giving us the ability to support

up to `400Gb` of outgoing traffic at the same time. As part of our traffic test, we wanted to attempt to check if WhiteCloud's DPDK network capabilities were able to maintain outbound network traffic consistently close to this upper limit.

Virtual Environment

For our tests, we needed to generate a large amount of packages to simulate heavy network traffic. To achieve this, our approach was to build multiple virtual machines inside the server, have them generate as much packages as possible and direct all the generated traffic outwards.

In order to count with the virtualization tools needed for our VM approach, `EPYC-05` was configured as a host in an WhiteCloud deployment, WhiteStack's own dockerized Openstack distribution. This cloud was comprised of four other hosts and a virtual machine. The VM was configured as the cloud's controller node, leaving all other hosts, and therefore `EPYC-05`, as purely compute nodes. Once the cloud was deployed `EPYC-05` counted with the Openstack services needed to build virtual machines and provide networking for them, deployed as docker containers.

To generate traffic, we built multiple VMs in the QEMU hypervisor provided by the Openstack compute services. These virtual machines were set up to run the `pktgen` DPDK tool, which is software specifically designed for DPDK traffic tests. `Pktgen` generates traffic and directs it to each of the VMs virtual interfaces, then we use OVS flows to direct all traffic from the VMs virtual interfaces to the server's physical interfaces. Finally, all traffic is sent to an external direction. A representation of the virtual environment is show on *Figure 2*.

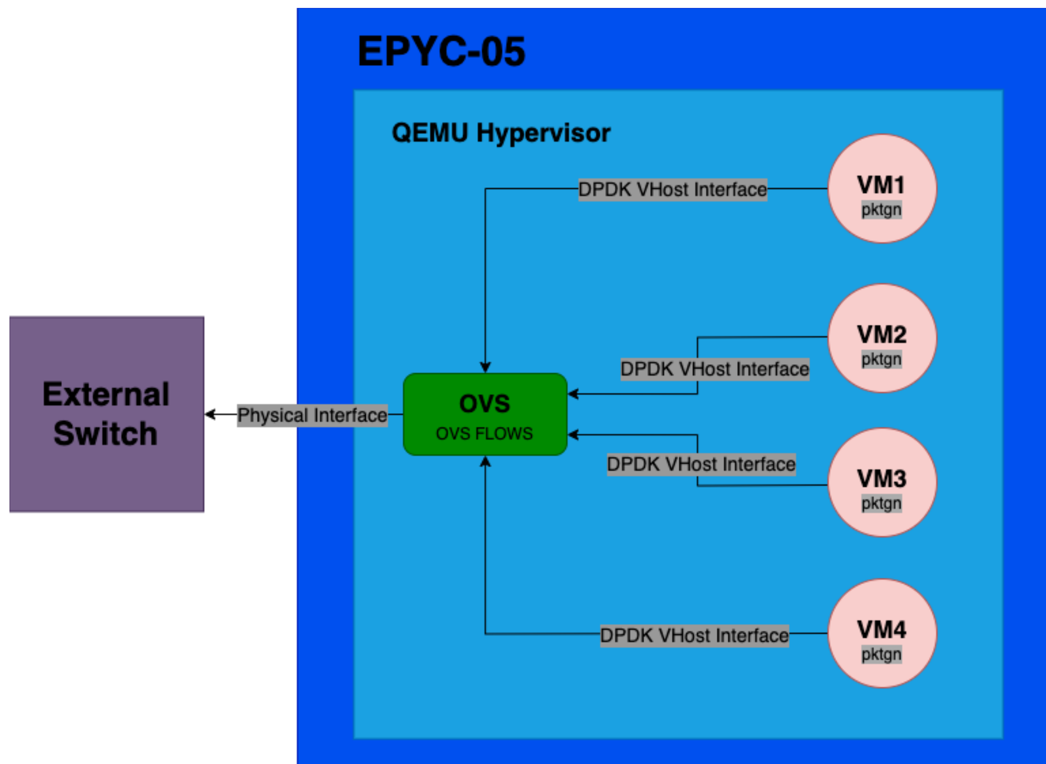
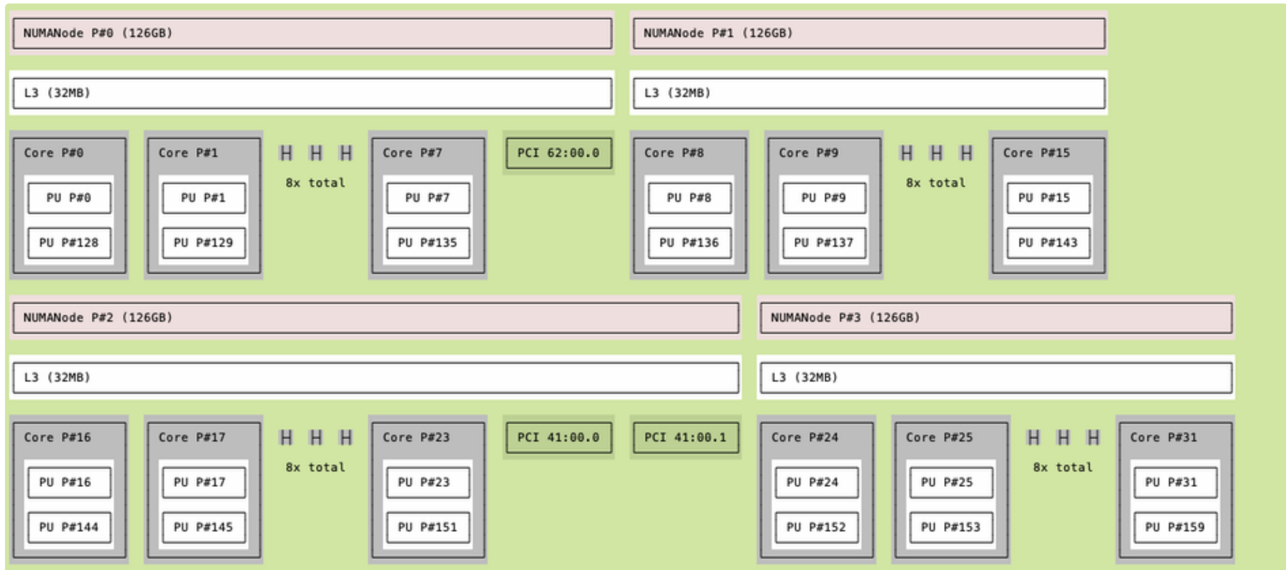


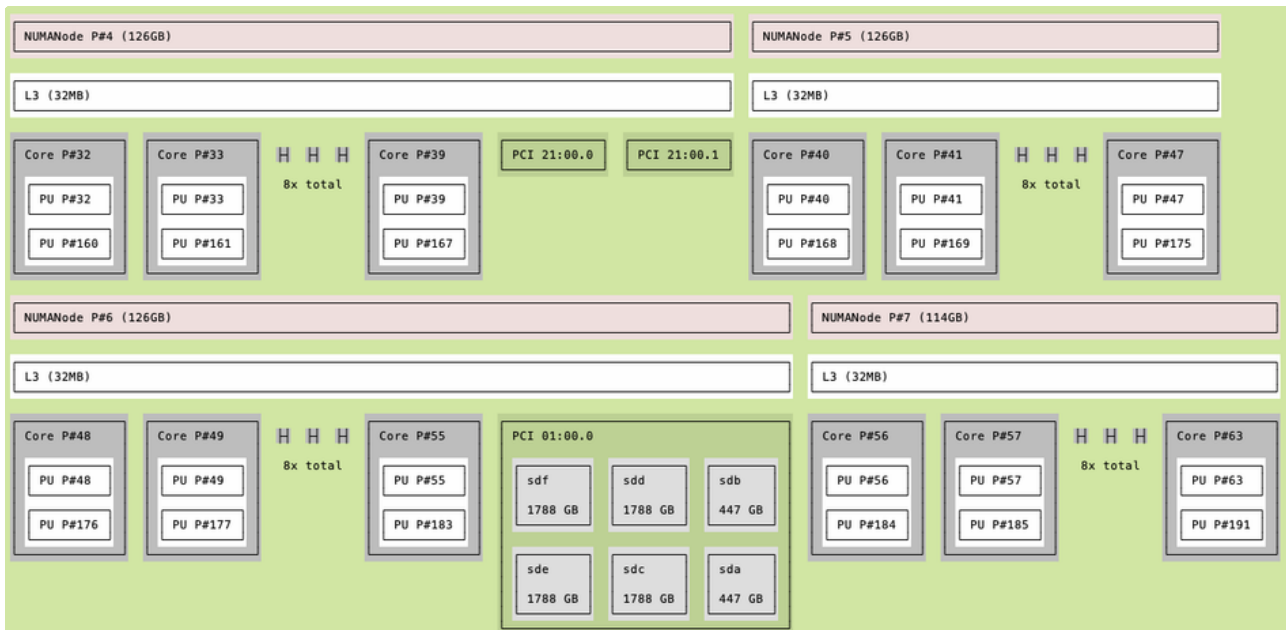
Figure 2 - Software Environment on EPYC-05

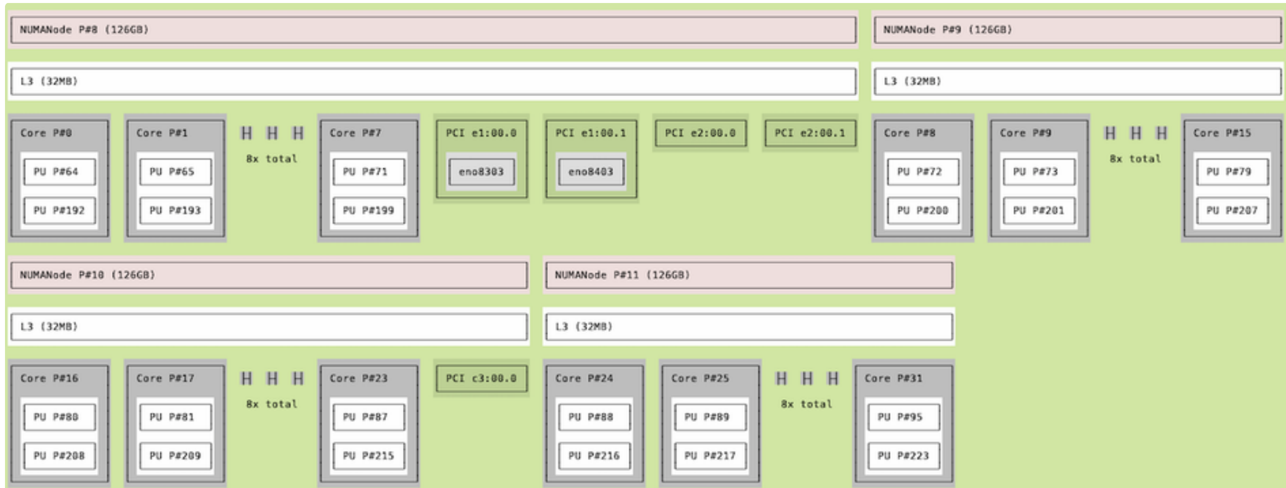
Configurations

EPYC-05 Bios Configurations

The server was configured with one NUMA per socket, resulting in sixteen NUMA groups matching the CCD groups. Each NUMA was conformed of eight physical cores, meaning, sixteen processing units. The resulting topology can be se seen in the diagram below.


NUMANode P#0 (126GB)
NUMANode P#1 (126GB)
L3 (32MB)
L3 (32MB)
Core P#0
PU P#0
PU P#128
Core P#1
PU P#1
PU P#129
8x total
Core P#7
PU P#7
PU P#135
PCI 62:00.0
Core P#8
PU P#8
PU P#136
Core P#9
PU P#9
PU P#137
8x total
Core P#15
PU P#15
PU P#143
NUMANode P#2 (126GB)
NUMANode P#3 (126GB)
L3 (32MB)
L3 (32MB)
Core P#16
PU P#16
PU P#144
Core P#17
PU P#17
PU P#145
8x total
Core P#23
PU P#23
PU P#151
PCI 41:00.0
PCI 41:00.1
Core P#24
PU P#24
PU P#152
Core P#25
PU P#25
PU P#153
8x total
Core P#31
PU P#31
PU P#159


NUMANode P#4 (126GB)
NUMANode P#5 (126GB)
L3 (32MB)
L3 (32MB)
Core P#32
PU P#32
PU P#160
Core P#33
PU P#33
PU P#161
8x total
Core P#39
PU P#39
PU P#167
PCI 21:00.0
PCI 21:00.1
Core P#40
PU P#40
PU P#168
Core P#41
PU P#41
PU P#169
8x total
Core P#47
PU P#47
PU P#175
NUMANode P#6 (126GB)
NUMANode P#7 (114GB)
L3 (32MB)
L3 (32MB)
Core P#48
PU P#48
PU P#176
Core P#49
PU P#49
PU P#177
8x total
Core P#55
PU P#55
PU P#183
PCI 01:00.0
sdf
1788 GB
sdd
1788 GB
sdb
447 GB
sde
1788 GB
sdc
1788 GB
sda
447 GB
Core P#56
PU P#56
PU P#184
Core P#57
PU P#57
PU P#185
8x total
Core P#63
PU P#63
PU P#191


NUMANode P#8 (126GB)
NUMANode P#9 (126GB)
L3 (32MB)
L3 (32MB)
Core P#0
PU P#64
PU P#192
Core P#1
PU P#65
PU P#193
8x total
Core P#7
PU P#71
PU P#199
PCI e1:00.0
eno8303
PCI e1:00.1
eno8403
PCI e2:00.0
PCI e2:00.1
Core P#8
PU P#72
PU P#200
Core P#9
PU P#73
PU P#201
8x total
Core P#15
PU P#79
PU P#207
NUMANode P#10 (126GB)
NUMANode P#11 (126GB)
L3 (32MB)
L3 (32MB)
Core P#16
PU P#80
PU P#208
Core P#17
PU P#81
PU P#209
8x total
Core P#23
PU P#87
PU P#215
PCI c3:00.0
Core P#24
PU P#88
PU P#216
Core P#25
PU P#89
PU P#217
8x total
Core P#31
PU P#95
PU P#223

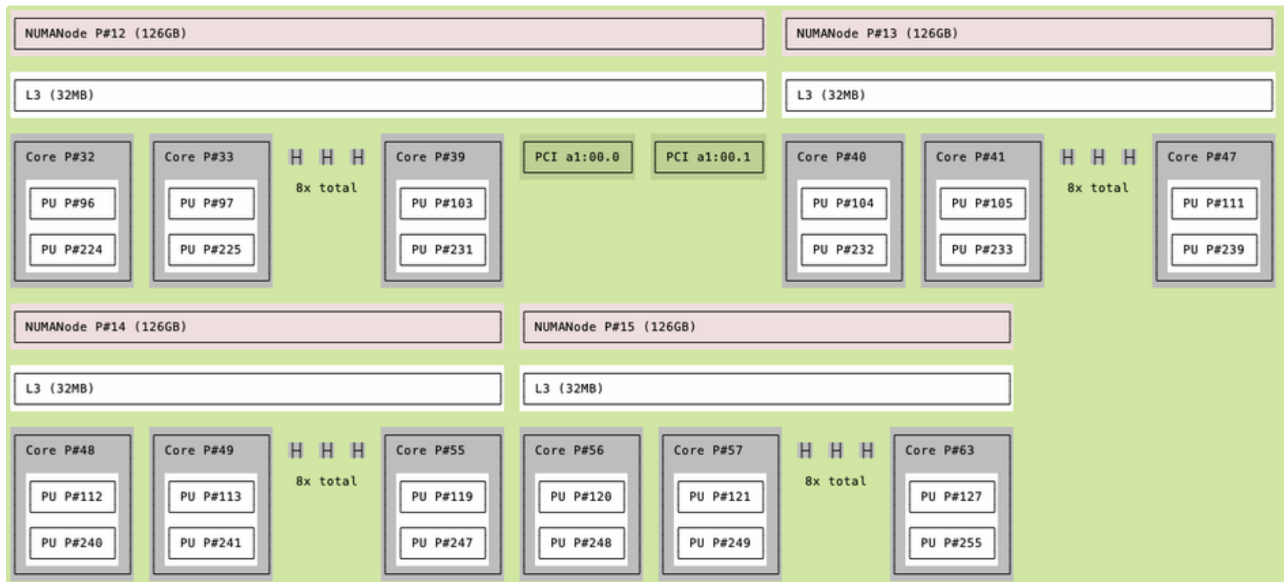


Figure 3 - Server Topology

In our tests, we decided to attempt saturating the port of a network adapter while only using cores from a single NUMA. We speculated that the generation and transportation of the amount of packages needed to reach the maximum data rate, could be handled successfully with eight vCPUs. Also, having all processes related to this test running on cores as close to each other as possible decreased any delays brought on by physical distance. We decided to use the NUMAs closest to the sockets where our `100Gb` network adapters were connected for similar reasons. To check the PCIs of the interfaces running on DPDK mode, we made use of the `dpdk-devbind.py` tool.

```
1  ~$ dpdk-devbind.py --status
2
3  Network devices using DPDK-compatible driver
4  ============================================
5  0000:21:00.0 'Ethernet Controller E810-C for QSFP 1592' drv=vfio-pci unused=ice,uio_pci_generic
6  0000:21:00.1 'Ethernet Controller E810-C for QSFP 1592' drv=vfio-pci unused=ice,uio_pci_generic
7  0000:41:00.0 'Ethernet Controller E810-C for QSFP 1592' drv=vfio-pci unused=ice,uio_pci_generic
8  0000:41:00.1 'Ethernet Controller E810-C for QSFP 1592' drv=vfio-pci unused=ice,uio_pci_generic
9  0000:a1:00.0 'Ethernet Controller E810-C for QSFP 1592' drv=vfio-pci unused=ice,uio_pci_generic
10 0000:a1:00.1 'Ethernet Controller E810-C for QSFP 1592' drv=vfio-pci unused=ice,uio_pci_generic
11 0000:e2:00.0 'Ethernet Controller E810-C for QSFP 1592' drv=vfio-pci unused=ice,uio_pci_generic
12 0000:e2:00.1 'Ethernet Controller E810-C for QSFP 1592' drv=vfio-pci unused=ice,uio_pci_generic
```

Crosschecking these PCIs with *Figure 3*, we could see that our tests will be focused on NUMAs 2, 3, 4, 5, 8, 9, 12 and 13.

These NUMAs were conformed by the following cores:

```
1 ~$ lscpu | grep "NUMA" | grep -e node2 -e node3 -e node4 -e node5 -e node8 -e node9 -e node12 -e node13
2 CCD node2 CPU(s):                  16-23,144-151
3 CCD node3 CPU(s):                  24-31,152-159
4 CCD node4 CPU(s):                  32-39,160-167
5 CCD node5 CPU(s):                  40-47,168-175
6 CCD node8 CPU(s):                  64-71,192-199
7 CCD node9 CPU(s):                  72-79,200-207
8 CCD node12 CPU(s):                 96-103,224-231
9 CCD node13 CPU(s):                 104-111,232-239
```

Grub Configs

As part of our tests, we needed to isolate some virtual CPUs in order to dedicate them to specific tasks. All vCPUs on the particular NUMAs were traffic would be going through, were isolated . Also, we configured hugepages for DPDK to use. This was done by editing the contents

of `/etc/default/grub` to the following:

```
~$ cat /etc/default/grub

GRUB_DEFAULT=0
GRUB_TIMEOUT_STYLE=hidden
GRUB_TIMEOUT=0
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT=""
GRUB_CMDLINE_LINUX="default_hugepagesz=1G hugepagesz=1G hugepages=1600 transparent_hugepage=never iommu=pt"
GRUB_CMDLINE_LINUX="$GRUB_CMDLINE_LINUX intel_iommu=on isolcpus=0,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,96,97,
98,99,100,101,102,103,104,105,106,107,108,109,110,111,144,145,146,147,148,149,150,151,152,153,154,155,156,
157,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,192,193,194,195,196,197,198,199,
200,201,202,203,204,205,206,207,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239
nohz=on nohz_full=0,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,
45,46,47,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,96,97,98,99,100,101,102,103,104,105,106,107,108,109,
110,111,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,168,
169,170,171,172,173,174,175,192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,224,225,226,227,
228,229,230,231,232,233,234,235,236,237,238,239 rcu_nocbs=0,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,96,97,98,99,
100,101,102,103,104,105,106,107,108,109,110,111,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,
159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,192,193,194,195,196,197,198,199,200,201,
202,203,204,205,206,207,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239 rcu_nocb_poll
numa_balancing=disable nmi_watchdog=1 audit=0 nosoftlockup hpet=disable tsc=reliable selinux=0
processor.max_cstate=1"
```

To apply these configurations, we ran the `update-grub` command and rebooted the server:

```
sudo update-grub
sudo reboot
```

WhiteCloud Configurations

For the tests, we configured WhiteCloud to deploy Openstack with DPDK networking. The general configurations for the WhiteCloud deployment were the following:

```
/etc/whitecloud$ cat cloud_configuration.yml
##
## Cloud Configuration
##

## Generic Configuration
neutron_plugin_agent: "openvswitch"
nova_console: "spice"
enable_neutron_provider_networks: yes

##
## Networks
##
# The network interface is the main "father" interface. All other interfaces inherit from it,
# if not changed explicitly.
network_interface: eno8303

#
# Neutron Network
# Used to provide external networking to neutron. Shouldn’t be exposed publicly to avoid connectivity issues.
neutron_external_interface: ens3
```

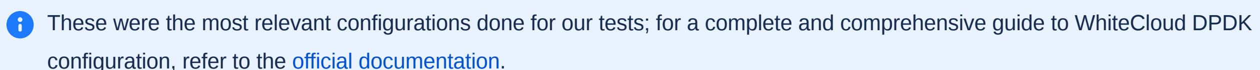
```
22
23 ##
24 ## VIPs and TLS
25 ##
26 # Internal VIP (mandatory)
27 # It needs to be a free IP address, belonging to the api_interface subnet
28 kolla_internal_vip_address: 10.100.44.4
29
30 enable_cinder: no
31 enable_grafana: yes
32 enable_prometheus: yes
33 enable_prometheus_server: yes
34 enable_prometheus_alertmanager: no
35 enable_prometheus_snmp_webhook: no
36 enable_ceilometer_prometheus_pushgateway: no
37
38 docker_disable_default_network: no
39 docker_disable_default_iptables_rules: no
40 docker_disable_ip_forward: no
```

The specific DPDK configurations needed for `EPYC-05` were specified in the inventory file:

```
1  /etc/whitecloud$ cat inventory.yml
2  ##
3  ## Cloud Inventory File
4  ##
5  roles:
6
7    control:
8      - dell-wc-lab-cont
9    compute:
10     - EPYC05
11   network:
12     - dell-wc-lab-cont
13   monitoring:
14     - dell-wc-lab-cont
15   storage:
16     - dell-wc-lab-cont
17   neutron:
18     - EPYC05
19
20 targets:
21   common:
22     ansible_become: yes
23   dell-wc-lab-cont:
24     network_interface: ens4
25     ansible_host: 10.100.41.216
26     ansible_user: ubuntu
27     host_needs_switching: no
28     ovsdpdk_extra_volumes: '{{ default_extra_volumes }}'
29
30   EPYC05:
31     ansible_host: 10.100.1.35
32     ansible_user: whitestack
33     ansible_become_pass: whitestack
34     enable_ovs_dpdk: yes
35     enable_openvswitch: yes
36     ovs_datapath: "netdev"
```

```
37    neutron_external_interface: "ens2f0,ens2f1,ens3f0,ens3f1,ens6f0,ens6f1,ens7f0,ens7f1"
38    neutron_bridge_name: "br-ex,br-ex2,br-ex3,br-ex4,br-ex5,br-ex6,br-ex7,br-ex8"
39    tunnel_interface: "br-ex"
40
41    ovs_dpdk_conf:
42      ovs:
43        bridge_mappings: "physnet1:br-ex,physnet2:br-ex2,physnet3:br-ex3,physnet4:br-ex4,physnet5:br-ex5,
44 physnet6:br-ex6,physnet7:br-ex7,physnet8:br-ex8"
45        port_mappings: "ens3f0:br-ex,ens3f1:br-ex2,ens2f0:br-ex3,ens2f1:br-ex4,ens7f0:br-ex5,ens7f1:br-ex6,
46 ens6f0:br-ex7,ens6f1:br-ex8"
47        cidr_mappings: "br-ex:10.100.4.63/24,br-ex2:10.100.5.63/24,br-ex3:10.100.6.63/24,br-ex4:10.100.7.63/24,
48 br-ex5:10.100.8.63/24,br-ex6:10.100.9.63/24,br-ex7:10.100.10.63/24,br-ex8:10.100.11.63/24"
49        ovs_cores: 16,144,24,152,32,160,40,168,64,192,72,200,96,224,107,232
50        pmd_cores: 21,22,23,29,30,31,37,38,39,45,46,47,69,70,71,77,76,79,101,102,103,109,110,111,145,146,147,
51 148,149,150,151,153,154,155,156,157,158,159,161,162,163,164,165,166,167,169,170,171,172,173,174,175,193,194,
52 195,196,197,198,199,201,202,203,204,205,206,207,225,226,227,228,229,230,231,233,234,235,236,237,238,239
53        dpdk_interface_driver: "vfio-pci"
54      interfaces:
55        ens2f0:
56          dpdk_interface_driver: "vfio-pci"
57          interface_driver: "ice"
58        ens2f1:
59          dpdk_interface_driver: "vfio-pci"
60          interface_driver: "ice"
61        ens3f0:
62          dpdk_interface_driver: "vfio-pci"
63          interface_driver: "ice"
64        ens3f1:
65          dpdk_interface_driver: "vfio-pci"
66          interface_driver: "ice"
67        ens6f0:
68          dpdk_interface_driver: "vfio-pci"
69          interface_driver: "ice"
70        ens6f1:
71          dpdk_interface_driver: "vfio-pci"
72          interface_driver: "ice"
73        ens7f0:
74          dpdk_interface_driver: "vfio-pci"
75          interface_driver: "ice"
76        ens7f1:
77          dpdk_interface_driver: "vfio-pci"
78          interface_driver: "ice"
```

As mentioned before, we specified to WhiteCloud which interfaces we wanted to configure and which DPDK compatible driver to use with them. We also specified which vCPUs of the server were going to be used as OVS and PMD cores. Additionally, we mapped each DPDK interface to a single bridge, with the objective to keep the traffic running through these interfaces separate.



These were the most relevant configurations done for our tests; for a complete and comprehensive guide to WhiteCloud DPDK configuration, refer to the official documentation.

Trafic Tests

After correctly configuring our servers and deploying our WhiteCloud, `EPYC-05` contained the Openstack docker containers related to the hosting and networking of virtual machines. The containers most relevant to our tests were the `nova_libvirt` and `ovsdpdk_vswitchd` containers. In order to have a greater control of the configurations for our tests, instead of managing the compute and network elements

through the Openstack CLI or WhiteCloud GUI, we used low level commands directly within these containers to create and configure virtual machines and interfaces.

Test Concept

We desired to test if we were able to saturate the outgoing traffic of an Intel Network adapter of an `100Gb` rate using the vCPUs from a singe CCD. Since we had four network adapters available, we wanted to use four CCDs. Each CCD had vCPUs dedicated to different tasks: four virtual CPUs for virtual machine computing, three vCPUs for DPDK vhost interfaces connected to the virtual machines, five vCPUs dedicated to PMD threads and two vCPUs dedicated to OVS. This resulted in the following core distribution:

CCD 0								CCD 1								CCD 2 (ens3f0)								CCD 3 (ens3f1)							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
CCD 4 (ens2f0)								CCD 5 (ens2f1)								CCD 6								CCD 7							
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191

CCD 8 (ens7f0)								CCD 9 (ens7f1)								CCD 10								CCD 11							
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
CCD 12 (ens6f0)								CCD 13 (ens6f1)								CCD 14								CCD 15							
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

OVS cores
PMD cores
DPDPK VHost cores
VM cores

Figure 4 - vCPU Distribution

Each CCD worked in tandem with a single port of the nearest adapter.

Consequently, we wanted to generate traffic inside the VM cores, using the `pktgen` tool, and direct said traffic to an external network through their DPDK vhost interfaces. To keep this process inside a single CCD, we kept the PMD threads of the DPDK vhost interfaces pinned to the vCPUs reserved for them, so that only these vCPUs managed the traffic of their respective interface. The same was done for the relevant OVS and PMD tasks, which were also pinned to the other cores in the same CCD.

`PKTGEN`

`Pktgen` is a tool compiled with DPDK which allows generating traffic from a specific core/vCPU, and directing it to a specific interface. This tool was used for our test in order to generate outbound traffic inside VMs. For our tests, we used an Ubuntu image that came prepared with PDKD and `pktgen`, but for references on how to build `pktgen` from scratch, refer to the official documentation.

Configuring the Test on WhiteCloud

First, we created the DPDK vhost interfaces, which provided connectivity to our VMs; we produced one interface for each DPDk vhost vCPU shown in *Figure 4.* The virtual interfaces for our VMs were named after their respective CCD and the VM vCPU producing traffic for it, i.e: `ccd3_vhost0` was the interface in `CCD3` that received traffic from the first vCPU of the VM in the same segment. This was achieved by running the following commands `ovs-vsctl` inside the `ovsdpdk_vswitchd` container:

```
1  ~$ sudo docker exec -it -u root ovsdpdk_vswitchd bash
2
3  ovs-vsctl --no-wait add-port br-ex2 ccd3_vhost0 -- set Interface ccd3_vhost0 type=dpdkvhostuser \
4  options:dpdk-devargs="class=eth,mac=33.22.33.44.33:00"
5  ovs-vsctl --no-wait add-port br-ex2 ccd3_vhost1 -- set Interface ccd3_vhost1 type=dpdkvhostuser \
6  options:dpdk-devargs="class=eth,mac=33.22.33.44.33:01"
7  ovs-vsctl --no-wait add-port br-ex2 ccd3_vhost2 -- set Interface ccd3_vhost2 type=dpdkvhostuser \
8  options:dpdk-devargs="class=eth,mac=33.22.33.44.33:02"
9
10 ovs-vsctl --no-wait add-port br-ex4 ccd5_vhost0 -- set Interface ccd5_vhost0 type=dpdkvhostuser \
```

```
11 options:dpdk-devargs="class=eth,mac=55.22.33.44.55:00"
12 ovs-vsctl --no-wait add-port br-ex4 ccd5_vhost1 -- set Interface ccd5_vhost1 type=dpdkvhostuser \
13 options:dpdk-devargs="class=eth,mac=55.22.33.44.55:01"
14 ovs-vsctl --no-wait add-port br-ex4 ccd5_vhost2 -- set Interface ccd5_vhost2 type=dpdkvhostuser \
15 options:dpdk-devargs="class=eth,mac=55.22.33.44.55:02"
16
17 ovs-vsctl --no-wait add-port br-ex6 ccd9_vhost0 -- set Interface ccd9_vhost0 type=dpdkvhostuser \
18 options:dpdk-devargs="class=eth,mac=99.22.33.44.99:00"
19 ovs-vsctl --no-wait add-port br-ex6 ccd9_vhost1 -- set Interface ccd9_vhost1 type=dpdkvhostuser \
20 options:dpdk-devargs="class=eth,mac=99.22.33.44.99:01"
21 ovs-vsctl --no-wait add-port br-ex6 ccd9_vhost2 -- set Interface ccd9_vhost2 type=dpdkvhostuser \
22 options:dpdk-devargs="class=eth,mac=99.22.33.44.99:02"
23
24 ovs-vsctl --no-wait add-port br-ex8 ccd13_vhost0 -- set Interface ccd13_vhost0 type=dpdkvhostuser \
25 options:dpdk-devargs="class=eth,mac=13.22.33.44.13:00"
26 ovs-vsctl --no-wait add-port br-ex8 ccd13_vhost1 -- set Interface ccd13_vhost1 type=dpdkvhostuser \
27 options:dpdk-devargs="class=eth,mac=13.22.33.44.13:01"
28 ovs-vsctl --no-wait add-port br-ex8 ccd13_vhost2 -- set Interface ccd13_vhost2 type=dpdkvhostuser \
29 options:dpdk-devargs="class=eth,mac=13.22.33.44.13:02"
```

Afterwards, we required that each DPDK vhost interface was pinned to its designated vCPU. To do so, we set the attribute `other_config:pmd-rxq-affinity` of each interface using the following commands:

```
1 ovs-vsctl --no-wait set Interface ccd3_vhost0 other_config:pmd-rxq-affinity="0:153"
2 ovs-vsctl --no-wait set Interface ccd3_vhost1 other_config:pmd-rxq-affinity="0:154"
3 ovs-vsctl --no-wait set Interface ccd3_vhost2 other_config:pmd-rxq-affinity="0:155"
4
5 ovs-vsctl --no-wait set Interface ccd5_vhost0 other_config:pmd-rxq-affinity="0:169"
6 ovs-vsctl --no-wait set Interface ccd5_vhost1 other_config:pmd-rxq-affinity="0:170"
7 ovs-vsctl --no-wait set Interface ccd5_vhost2 other_config:pmd-rxq-affinity="0:171"
8
9 ovs-vsctl --no-wait set Interface ccd9_vhost0 other_config:pmd-rxq-affinity="0:201"
10 ovs-vsctl --no-wait set Interface ccd9_vhost1 other_config:pmd-rxq-affinity="0:202"
11 ovs-vsctl --no-wait set Interface ccd9_vhost2 other_config:pmd-rxq-affinity="0:203"
12
13 ovs-vsctl --no-wait set Interface ccd13_vhost0 other_config:pmd-rxq-affinity="0:233"
14 ovs-vsctl --no-wait set Interface ccd13_vhost1 other_config:pmd-rxq-affinity="0:234"
15 ovs-vsctl --no-wait set Interface ccd13_vhost2 other_config:pmd-rxq-affinity="0:235"
```

Additionally, we applied the following DPDK configurations to OVS:

```
1 ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-limit=20480
2 ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem=20480,0
3
4 ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-extra="-n 8
5 -a 0000:21:00.0,mprq_en=1,mprq_log_stride_num=9,txq_inline_mpw=128,rxq_pkt_pad_en=1,mprq_tstore_memcpy=1
6 -a 0000:21:00.1,mprq_en=1,mprq_log_stride_num=9,txq_inline_mpw=128,rxq_pkt_pad_en=1,mprq_tstore_memcpy=1
7 -a 0000:41:00.0,mprq_en=1,mprq_log_stride_num=9,txq_inline_mpw=128,rxq_pkt_pad_en=1,mprq_tstore_memcpy=1
8 -a 0000:41:00.1,mprq_en=1,mprq_log_stride_num=9,txq_inline_mpw=128,rxq_pkt_pad_en=1,mprq_tstore_memcpy=1
9 -a 0000:a1:00.0,mprq_en=1,mprq_log_stride_num=9,txq_inline_mpw=128,rxq_pkt_pad_en=1,mprq_tstore_memcpy=1
10 -a 0000:a1:00.1,mprq_en=1,mprq_log_stride_num=9,txq_inline_mpw=128,rxq_pkt_pad_en=1,mprq_tstore_memcpy=1
11 -a 0000:e2:00.0,mprq_en=1,mprq_log_stride_num=9,txq_inline_mpw=128,rxq_pkt_pad_en=1,mprq_tstore_memcpy=1
12 -a 0000:e2:00.1,mprq_en=1,mprq_log_stride_num=9,txq_inline_mpw=128,rxq_pkt_pad_en=1,mprq_tstore_memcpy=1"
```

Next, we configured our physical interfaces and pinned their queues to their designated cores inside their respective CCDs:

```
1 ovs-vsctl --no-wait set Interface ens2f1 options:n_txq_desc=4096
2 ovs-vsctl --no-wait set Interface ens2f1 options:n_rxq_desc=4096
3 ovs-vsctl --no-wait set Interface ens3f1 options:n_txq_desc=4096
```

```
4  ovs-vsctl --no-wait set Interface ens3f1 options:n_rxq_desc=4096
5  ovs-vsctl --no-wait set interface ens2f1 options:n_rxq=2
6  ovs-vsctl --no-wait set interface ens3f1 options:n_rxq=2
7
8  ovs-vsctl --no-wait set Interface ens6f1 options:n_txq_desc=4096
9  ovs-vsctl --no-wait set Interface ens6f1 options:n_rxq_desc=4096
10 ovs-vsctl --no-wait set Interface ens7f1 options:n_txq_desc=4096
11 ovs-vsctl --no-wait set Interface ens7f1 options:n_rxq_desc=4096
12 ovs-vsctl --no-wait set interface ens6f1 options:n_rxq=2
13 ovs-vsctl --no-wait set interface ens7f1 options:n_rxq=2
14
15 ovs-vsctl --no-wait set Interface ens2f1 other_config:pmd-rxq-affinity="0:45,1:46"
16 ovs-vsctl --no-wait set Interface ens3f1 other_config:pmd-rxq-affinity="0:29,1:30"
17 ovs-vsctl --no-wait set Interface ens6f1 other_config:pmd-rxq-affinity="0:109,1:110"
18 ovs-vsctl --no-wait set Interface ens7f1 other_config:pmd-rxq-affinity="0:77,1:78"
```

Once our interfaces were created and configured, it was necessary to restart OVS to apply the changes. This is done in WhiteCloud by running the deployer with the action: `--action restart-ovs`. Once restarted, we needed to add OVS flows that linked the traffic received by the DPDK vhost interfaces to the physical interfaces in the same CCD, by running the following commands inside the `ovsdpdk_vswitchd` container:

```
1  ovs-vsctl set Interface ens2f1 ofport_request=2001
2  ovs-vsctl set Interface ens3f1 ofport_request=3001
3  ovs-vsctl set Interface ens6f1 ofport_request=6001
4  ovs-vsctl set Interface ens7f1 ofport_request=7001
5
6  ovs-ofctl add-flow br-ex2 in_port=ccd3_vhost0,dl_type=0x800,idle_timeout=0,actions=output:3001
7  ovs-ofctl add-flow br-ex2 in_port=ccd3_vhost1,dl_type=0x800,idle_timeout=0,actions=output:3001
8  ovs-ofctl add-flow br-ex2 in_port=ccd3_vhost2,dl_type=0x800,idle_timeout=0,actions=output:3001
9
10 ovs-ofctl add-flow br-ex4 in_port=ccd5_vhost0,dl_type=0x800,idle_timeout=0,actions=output:2001
11 ovs-ofctl add-flow br-ex4 in_port=ccd5_vhost1,dl_type=0x800,idle_timeout=0,actions=output:2001
12 ovs-ofctl add-flow br-ex4 in_port=ccd5_vhost2,dl_type=0x800,idle_timeout=0,actions=output:2001
13
14 ovs-ofctl add-flow br-ex6 in_port=ccd9_vhost0,dl_type=0x800,idle_timeout=0,actions=output:7001
15 ovs-ofctl add-flow br-ex6 in_port=ccd9_vhost1,dl_type=0x800,idle_timeout=0,actions=output:7001
16 ovs-ofctl add-flow br-ex6 in_port=ccd9_vhost2,dl_type=0x800,idle_timeout=0,actions=output:7001
17
18 ovs-ofctl add-flow br-ex8 in_port=ccd13_vhost0,dl_type=0x800,idle_timeout=0,actions=output:6001
19 ovs-ofctl add-flow br-ex8 in_port=ccd13_vhost1,dl_type=0x800,idle_timeout=0,actions=output:6001
20 ovs-ofctl add-flow br-ex8 in_port=ccd13_vhost2,dl_type=0x800,idle_timeout=0,actions=output:6001
```

When the networking was done, we built virtual machines using a custom Ubuntu image, which had all the tools needed for our pre-installed tests, and saved this image in this repository. We uploaded this image to the respective container use the `docker cp` command:

```
1 sudo docker cp ubuntu20_AMD_Tests.iso nova_libvirt:/
```

Then, inside the `nova_libvirt` container, we placed the image in a specific directory and made copies for each virtual machine we wished to build:

```
1 ~ $ sudo docker exec -it -u root nova_libvirt bash
2 ~ $mkdir /amd_tests
3 ~ $mv ubuntu20_AMD_Tests.iso /amd_tests/
4 ~ $cd amd_tests
5 ~ $cp ubuntu20_AMD_Tests.iso ubuntu20_AMD_Tests2.iso
6 ~ $cp ubuntu20_AMD_Tests.iso ubuntu20_AMD_Tests4.iso
7 ~ $cp ubuntu20_AMD_Tests.iso ubuntu20_AMD_Tests6.iso
8 ~ $cp ubuntu20_AMD_Tests.iso ubuntu20_AMD_Tests8.iso
```

⚠ These images are quite heavy, coming in at 5.7Gb for each. QEMU requires a separate image for each VM, so check your disk space before generating the copies.

Next, we created VMs inside `nova_libvirt` using these images and their respective DPDK vhost interfaces. Each VM only had interfaces that were pinned inside the same CCD. This was done with the `/usr/bin/qemu-system-x86_64` tool, which came included in the `nova_libvirt` container:

```
# VM CCD 3
/usr/bin/qemu-system-x86_64 -cpu host -smp cores=4 --enable-kvm -hda /amd_tests/ubuntu20_AMD_Tests2.iso \
-m 8192 -chardev socket,id=char1,path=/run/openvswitch/ccd3_vhost0 \
-netdev type=vhost-user,id=mynet1,chardev=char1,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:01,netdev=mynet1,mq=on,vectors=4,mrg_rxbuf=off \
-chardev socket,id=char11,path=/run/openvswitch/ccd3_vhost1 \
-netdev type=vhost-user,id=mynet11,chardev=char11,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:11,netdev=mynet11,mq=on,vectors=4,mrg_rxbuf=off \
-chardev socket,id=char2,path=/run/openvswitch/ccd3_vhost2 \
-netdev type=vhost-user,id=mynet2,chardev=char2,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:02,netdev=mynet2,mq=on,vectors=4,mrg_rxbuf=off \
-object memory-backend-file,id=mem,size=8G,mem-path=/dev/hugepages,share=on -numa node,memdev=mem \
-mem-prealloc -nic user,id=vmnic,hostfwd=tcp:127.0.0.1:5553-:22 -monitor telnet:127.0.0.1:55557,server,nowait \
-serial telnet:127.0.0.1:55558,server,nowait -name vm1,debug-threads=on --daemonize
# VM CCD 5
/usr/bin/qemu-system-x86_64 -cpu host -smp cores=4 --enable-kvm -hda /amd_tests/ubuntu20_AMD_Tests4.iso \
-m 8192 -chardev socket,id=char1,path=/run/openvswitch/ccd5_vhost0 \
-netdev type=vhost-user,id=mynet1,chardev=char1,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:01,netdev=mynet1,mq=on,vectors=4,mrg_rxbuf=off \
-chardev socket,id=char11,path=/run/openvswitch/ccd5_vhost1 \
-netdev type=vhost-user,id=mynet11,chardev=char11,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:11,netdev=mynet11,mq=on,vectors=4,mrg_rxbuf=off \
-chardev socket,id=char2,path=/run/openvswitch/ccd5_vhost2 \
-netdev type=vhost-user,id=mynet2,chardev=char2,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:02,netdev=mynet2,mq=on,vectors=4,mrg_rxbuf=off \
-object memory-backend-file,id=mem,size=8G,mem-path=/dev/hugepages,share=on -numa node,memdev=mem \
-mem-prealloc -nic user,id=vmnic,hostfwd=tcp:127.0.0.1:5555-:22 -monitor telnet:127.0.0.1:55561,server,nowait \
-serial telnet:127.0.0.1:55562,server,nowait -name vm3,debug-threads=on --daemonize
# VM CCD 9
/usr/bin/qemu-system-x86_64 -cpu host -smp cores=4 --enable-kvm -hda /amd_tests/ubuntu20_AMD_Tests6.iso \
-m 8192 -chardev socket,id=char1,path=/run/openvswitch/ccd9_vhost0 \
-netdev type=vhost-user,id=mynet1,chardev=char1,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:01,netdev=mynet1,mq=on,vectors=4,mrg_rxbuf=off \
-chardev socket,id=char11,path=/run/openvswitch/ccd9_vhost1 \
-netdev type=vhost-user,id=mynet11,chardev=char11,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:11,netdev=mynet11,mq=on,vectors=4,mrg_rxbuf=off \
-chardev socket,id=char2,path=/run/openvswitch/ccd9_vhost2 \
-netdev type=vhost-user,id=mynet2,chardev=char2,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:02,netdev=mynet2,mq=on,vectors=4,mrg_rxbuf=off \
-object memory-backend-file,id=mem,size=8G,mem-path=/dev/hugepages,share=on -numa node,memdev=mem\
-mem-prealloc -nic user,id=vmnic,hostfwd=tcp:127.0.0.1:5557-:22 -monitor telnet:127.0.0.1:55565,server,nowait \
-serial telnet:127.0.0.1:55566,server,nowait -name vm5,debug-threads=on --daemonize
# VM CCD 13
/usr/bin/qemu-system-x86_64 -cpu host -smp cores=4 --enable-kvm -hda /amd_tests/ubuntu20_AMD_Tests8.iso \
-m 8192 -chardev socket,id=char1,path=/run/openvswitch/ccd13_vhost0 \
-netdev type=vhost-user,id=mynet1,chardev=char1,vhostforce,queues=1 -device\
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:01,netdev=mynet1,mq=on,vectors=4,mrg_rxbuf=off \
-chardev socket,id=char11,path=/run/openvswitch/ccd13_vhost1 \
-netdev type=vhost-user,id=mynet11,chardev=char11,vhostforce,queues=1 -device \
virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:11,netdev=mynet11,mq=on,vectors=4,mrg_rxbuf=off \
```

```
51 -chardev socket,id=char2,path=/run/openvswitch/ccd13_vhost2 \
52 -netdev type=vhost-user,id=mynet2,chardev=char2,vhostforce,queues=1 -device \
53 virtio-net-pci,speed=100000,duplex=full,mac=52:55:00:d1:55:02,netdev=mynet2,mq=on,vectors=4,mrg_rxbuf=off \
54 -object memory-backend-file,id=mem,size=8G,mem-path=/dev/hugepages,share=on -numa node,memdev=mem \
55 -mem-prealloc -nic user,id=vmnic,hostfwd=tcp:127.0.0.1:5559-:22 -monitor telnet:127.0.0.1:55569,server,nowait \
56 -serial telnet:127.0.0.1:55570,server,nowait -name vm7,debug-threads=on --daemonize
```

One important observation in these commands is the `-serial` option. This option allowed us to open a port in the local direction and bind it to the VM console. This port lets us access the VMs via a `telnet` command. For example: for the first VM the command contains `-serial telnet:127.0.0.1:55558,server,nowait`: this means that to access this VM we used the command `telnet localhost 55556`. By default, our image had the `root:root` user configured for access.

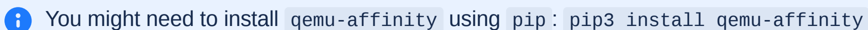
You might need to install `telnet` using `apt-get`:

`sudo apt-get update`

`sudo apt-get install telnet`

```
1  # telnet localhost 55558
2  Trying 127.0.0.1...
3  Connected to localhost.
4  Escape character is '^]'.
5
6  Ubuntu 18.04.3 LTS test-dpdk ttyS0
7
8  test-dpdk login: root
9  Password:
10
11 root@test-dpdk:~#
```

Once the VMs were up and running, we pinned their processes to the cores we had reserved for them. These cores had to be on the same CCD on which their interface are pinned. We achieved this pinning through the `quemu-affinity` tool.

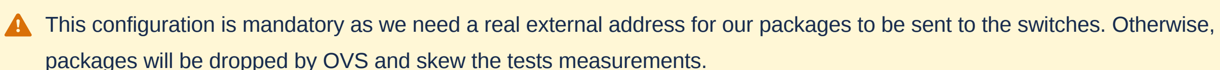


You might need to install `qemu-affinity` using `pip`: `pip3 install qemu-affinity`

```
1 qemu-affinity -v -k 17 18 19 20 -- $(ps -aux | grep -e qemu | grep -e vm0 | grep -v 'grep' | awk '{print $2}')
2 qemu-affinity -v -k 25 26 27 28 -- $(ps -aux | grep -e qemu | grep -e vm1 | grep -v 'grep' | awk '{print $2}')
3 qemu-affinity -v -k 33 34 35 36 -- $(ps -aux | grep -e qemu | grep -e vm2 | grep -v 'grep' | awk '{print $2}')
4 qemu-affinity -v -k 41 42 43 44 -- $(ps -aux | grep -e qemu | grep -e vm3 | grep -v 'grep' | awk '{print $2}')
5 qemu-affinity -v -k 65 66 67 68 -- $(ps -aux | grep -e qemu | grep -e vm4 | grep -v 'grep' | awk '{print $2}')
6 qemu-affinity -v -k 73 74 75 76 -- $(ps -aux | grep -e qemu | grep -e vm5 | grep -v 'grep' | awk '{print $2}')
7 qemu-affinity -v -k 97 98 99 100 -- $(ps -aux | grep -e qemu | grep -e vm6 | grep -v 'grep' | awk '{print $2}')
8 qemu-affinity -v -k 105 106 107 108 -- $(ps -aux | grep -e qemu | grep -e vm7 | grep -v 'grep' | awk '{print $2}'
```

Finally, necessary to run our tests, we needed to configure a new static MAC direction on the Dell switches MAC address table, considering we had to configure the generated packages with a real MAC destination, so that the generated traffic was properly directed to the switches. To achieve this, we ran these commands on both switches:

```
1 configure terminal
2 mac-address-table static 52:55:00:d1:55:04 vlan 1 interface ethernet 1/1/4
3 end
```



This configuration is mandatory as we need a real external address for our packages to be sent to the switches. Otherwise, packages will be dropped by OVS and skew the tests measurements.

Running the Tests | Pktgen

Once our VMs were up and running using our DPDK vhost interfaces, the flows in OVS provided connectivity to the physical interfaces and everything was pinned to its designated vCPUs, we were ready to start generating traffic inside the VMs and measure the amount of packages being output.

First, we had to enter each of the VMs and carry out the following commands:

1. Bind every vhost interface into DPDK mode:

```
1 sudo ./dpdk-devbind.py -b igb_uio 00:04.0
2 sudo ./dpdk-devbind.py -b igb_uio 00:05.0
3 sudo ./dpdk-devbind.py -b igb_uio 00:06.0
```

2. Set up hugepages for `pktgen`:

```
1 dpdk-hugepages.py --setup 4G
```

3. Start pktgen:

```
1 pktgen -l 0,1,2,3 --main-lcore 3 -a 00:04.0 -a 00:05.0 -a 00:06.0  -- -P -m "0.0,1.1,2.2"
```

   Here we can see that we assigned one vCPU to each interface, leaving one vCPU to be the `main-lcore`.

Once this commands were run, we were left with the four VMs showing the `pktgen` console:

```
 1 \ Ports 0-2 of 3   <Main Page>  Copyright(c) <2010-2021>, Intel Corporation
 2   Flags:Port        : P------Sngl       :0 P------Sngl       :1 P------Sngl       :2
 3 Link State          :       <UP-100000-FD>       <UP-100000-FD>       <UP-100000-FD>     ---Total Rate---
 4 Pkts/s Rx           :                    0                    0                    0                    0
 5        Tx           :                    0                    0                    0                    0
 6 MBits/s Rx/Tx       :                  0/0                  0/0                  0/0                  0/0
 7 Pkts/s Rx Max       :                    1                    1                    1                    3
 8        Tx Max       :                    0                    0                    0                    0
 9 Broadcast           :                    0                    0                    0
10 Multicast           :                    1                    2                    3
11 Sizes 64            :                    0                    0                    0
12       65-127        :                    0                    0                    0
13       128-255       :                    0                    0                    0
14       256-511       :                    1                    2                    3
15       512-1023      :                    0                    0                    0
16       1024-1518     :                    0                    0                    0
17 Runts/Jumbos        :                  0/0                  0/0                  0/0
18 ARP/ICMP Pkts       :                  0/0                  0/0                  0/0
19 Errors Rx/Tx        :                  0/0                  0/0                  0/0
20 Total Rx Pkts       :                    1                    1                    1
21       Tx Pkts       :                    0                    0                    0
22       Rx/Tx MBs     :                  0/0                  0/0                  0/0
23 TCP Flags           :               .A....               .A....               .A....
24 TCP Seq/Ack         :  305419896/305419920  305419896/305419920  305419896/305419920
25 Pattern Type        :              abcd...              abcd...              abcd...
26 Tx Count/% Rate     :        Forever /100%        Forever /100%        Forever /100%
27 Pkt Size/Tx Burst   :            64 /  128            64 /  128            64 /  128
28 TTL/Port Src/Dest   :       64/ 1234/ 5678       64/ 1234/ 5678       64/ 1234/ 5678
29 Pkt Type:VLAN ID    :      IPv4 / TCP:0001      IPv4 / TCP:0001      IPv4 / TCP:0001
30 802.1p CoS/DSCP/IPP :            0/   0/  0            0/   0/  0            0/   0/  0
31 VxLAN Flg/Grp/vid   :     0000/    0/    0     0000/    0/    0     0000/    0/    0
32 IP  Destination     :          192.168.1.1          192.168.0.1          192.168.3.1
33     Source          :       192.168.0.1/24       192.168.1.1/24       192.168.2.1/24
34 MAC Destination     :    52:55:00:d1:55:11    52:55:00:d1:55:01    00:00:00:00:00:00
```

```
35     Source          :    52:55:00:d1:55:01    52:55:00:d1:55:11    52:55:00:d1:55:02
36 PCI Vendor/Addr     :    1af4:1000/00:04.0    1af4:1000/00:05.0    1af4:1000/00:06.0
37 -- Pktgen 22.04.1 (DPDK 23.03.0-rc0)  Powered by DPDK  (pid:2158) -------------
38 ** Version: DPDK 23.03.0-rc0, Command Line Interface without timers
39 Pktgen:/>
```

Before we started generating traffic, we had to configure `pktgen` so that it generated the class of packages that best served our needs. We needed to test UPD packages and wished to set their MTU to `1500`, to make it easier to saturate the interfaces while using less packages. Finally, we wanted to set the destination of these packages to the MAC address we previously added to our switches' MAC address-table: `52:55:00:d1:55:04`. All these configuration were applied with the following commands on all VMs:

```
1 set all proto udp
2 set all size 1500
3 set all dst mac 52:55:00:d1:55:04
```

We confirmed these configurations by verifying that the respective info had been updated on the `pktgen` console:

```
1  \ Ports 0-2 of 3   <Main Page>  Copyright(c) <2010-2021>, Intel Corporation
2    Flags:Port        : P------Sngl       :0 P------Sngl       :1 P------Sngl       :2
3  Link State          :       <UP-100000-FD>       <UP-100000-FD>       <UP-100000-FD>     ---Total Rate---
4  Pkts/s Rx           :                    0                    0                    0                    0
5         Tx           :                    0                    0                    0                    0
6  MBits/s Rx/Tx       :                  0/0                  0/0                  0/0                  0/0
7  Pkts/s Rx Max       :                    1                    1                    1                    3
8         Tx Max       :                    0                    0                    0                    0
9  Broadcast           :                    0                    0                    0
10 Multicast           :                   19                   38                   57
11 Sizes 64            :                    0                    0                    0
12       65-127        :                    1                    2                    3
13       128-255       :                    0                    0                    0
14       256-511       :                   19                   38                   57
15       512-1023      :                    0                    0                    0
16       1024-1518     :                    0                    0                    0
17 Runts/Jumbos        :                  0/0                  0/0                  0/0
18 ARP/ICMP Pkts       :                  0/0                  0/0                  0/0
19 Errors Rx/Tx        :                  0/0                  0/0                  0/0
20 Total Rx Pkts       :                   20                   20                   20
21       Tx Pkts       :                    0                    0                    0
22       Rx/Tx MBs     :                  0/0                  0/0                  0/0
23 TCP Flags           :                .A....               .A....               .A....
24 TCP Seq/Ack         :  305419896/305419920  305419896/305419920  305419896/305419920
25 Pattern Type        :              abcd...              abcd...              abcd...
26 Tx Count/% Rate     :        Forever /100%        Forever /100%        Forever /100%
27 Pkt Size/Tx Burst   :          1500 /  128          1500 /  128          1500 /  128         # <------ MTU 1500
28 TTL/Port Src/Dest   :       64/ 1234/ 5678       64/ 1234/ 5678       64/ 1234/ 5678
29 Pkt Type:VLAN ID    :      IPv4 / UDP:0001      IPv4 / UDP:0001      IPv4 / UDP:0001         # <------ UPD PROTO
30 802.1p CoS/DSCP/IPP :            0/  0/  0            0/  0/  0            0/  0/  0
31 VxLAN Flg/Grp/vid   :     0000/    0/    0     0000/    0/    0     0000/    0/    0
32 IP  Destination     :          192.168.1.1          192.168.0.1          192.168.3.1
33     Source          :       192.168.0.1/24       192.168.1.1/24       192.168.2.1/24
34 MAC Destination     :    52:55:00:d1:55:04    52:55:00:d1:55:04    52:55:00:d1:55:04         # <------ MAC DST
35     Source          :    52:55:00:d1:55:01    52:55:00:d1:55:11    52:55:00:d1:55:02
36 PCI Vendor/Addr     :    1af4:1000/00:04.0    1af4:1000/00:05.0    1af4:1000/00:06.0
37 -- Pktgen 22.04.1 (DPDK 23.03.0-rc0)  Powered by DPDK  (pid:2158) -------------
38 ** Version: DPDK 23.03.0-rc0, Command Line Interface without timers
39 Pktgen:/>
```

After all these configurations were executed, we were ready to start generating traffic from the VMs. Having every vCPU start generating traffic at the same time proved to be a strenuous action for OVS, as we would observe packages being dropped in this case. This meant that we had to start generating traffic one vCPU at a time, waiting for a minute before moving on to the next vCPU. This was done in the following order:

> 1. Enter a VM using `telnet` and launch `pktgen`.
>
>   a. Start traffic on the first core, then wait for a minute: `start 0`
>
>   b. Repeat for other cores with `start 1`, and `start 2`
>
> 2. Repeat for the rest of the VMs

Another important observation is that, on CCDs 9 and 13, greater traffic was achieved when generating traffic from just two cores than from three. The reasons for this behavior are still being studied, but we theorize that some cores must have a better performance than others and that generating traffic from an under-performing core might be detrimental to the total result.

Once we completed these steps, we had successfully started generating traffic from our VMs. At this point, we used WhiteCloud's own Prometheus server, an open source monitoring system included with our distribution, to observe and record the traffic transiting through each interface. These results will be presented in the following section.

Results

First, we can see in the following graph how the traffic in each interface evolved while starting the generation in each vCPU:

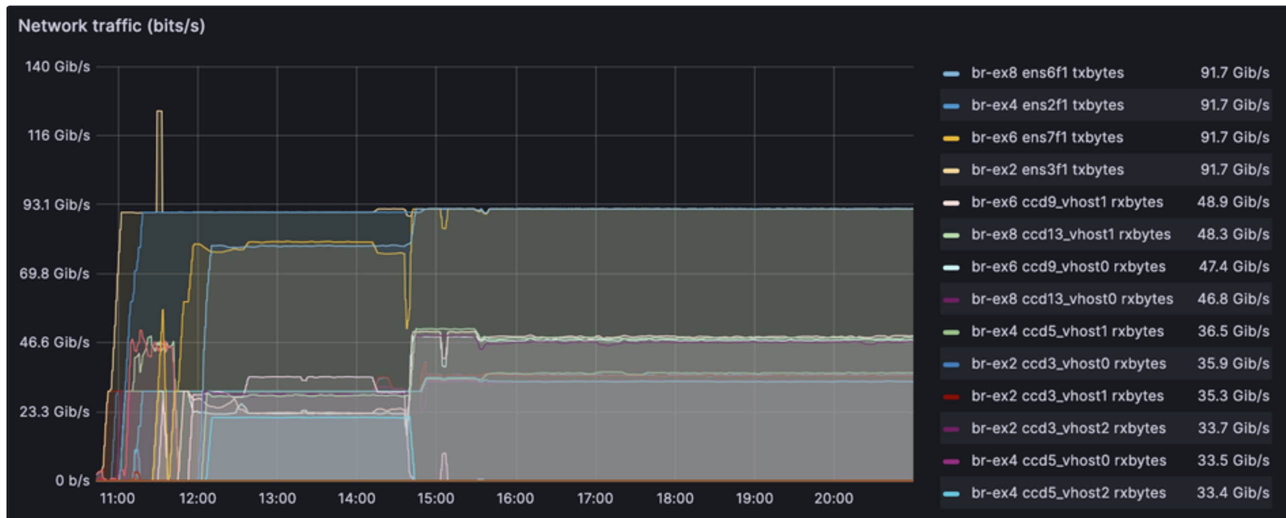


Figure 5 - Network Traffic (Gib/s)

Figure 5 shows us the traffic running through each interface in our lab's environment. At the beginning of the test we started generating traffic gradually until every vCPU was generating traffic for its interface, between `10:00` and `12:30` on the graph. At this point, we had two of our physical interfaces near saturation (`~90Bib/s`), while the other two maintained less traffic in an inconsistent manner. From this point up until `14:30`, we tweaked the tests configurations in order to achieve greater traffic, both on each interface and as a total. This is were most of the previously exposed configurations were settled on.

A turning point can be seen at `14:30`, when we discovered that certain vCPUS generating traffic were negatively affecting the results. The moment traffic generation was stopped for these vCPUs, the traffic in their respective physical interfaces increased and matched the traffic achieved by the other two interfaces initially. After these changes were made and enough time was given for the traffic to stabilize, we saw that our DPDK dataplane was able to transport traffic from the DPDK vhost interfaces to the physical interfaces at a rate nearing the maximum allowed by the network adapters.

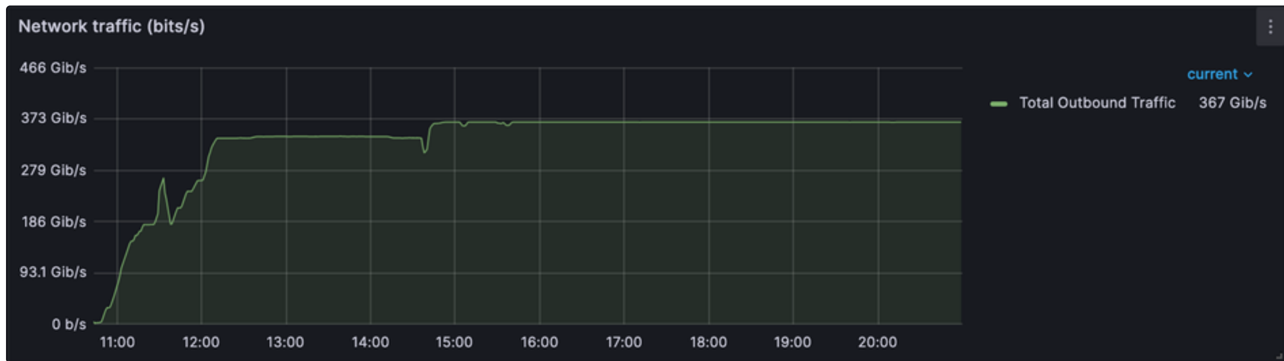


Figure 6 - Total Outbound Network Traffic

Each physical interface maintains a traffic of `91.7Gib/s`, meaning, the DPDK dataplane is able to handle an outgoing traffic of a total of `367Gib/s`, nearly `92%` of the maximum allowed by the network adapters. The total outbound traffic and its evolution during the test can be seen above on *Figure 6.*

For a second test, we wished to understand how OVS would handle traffic of different MTU. To do so, we repeated the previous experiment, but on this occasion we configured `pktgen` to generate packages of MTU `64`.After starting traffic generation, and waiting for traffic to stabilize, we increased the MTU to 1500. Again, we waited for traffic to stabilize before lowering the MTU one last time to 64. The behavior of the traffic across interfaces during this test can be seen in the graph below.

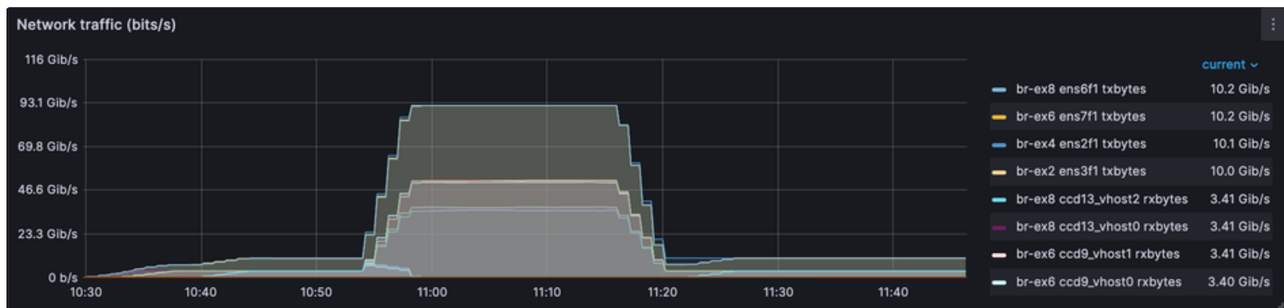


Figure 7 - Network Traffic MTU Test (Gib/s)

As expected, *Figure 7* shows how traffic comprised of MTU `64` packages stabilizes on each physical interface at approximately `10Gib/s`. Traffic then spikes when package size is increased to `1500`, reaching the previously achieved volume of traffic. Here we note again that in order to achieve this amount of traffic, traffic generation had to be stop on certain vCPUs. Finally, when returning packages to their original MTU, we see traffic returning too to its original condition. Whats is interesting to observe is then the amount of individual packages being transported, instead of the total size of traffic. This is shown on the graph below.

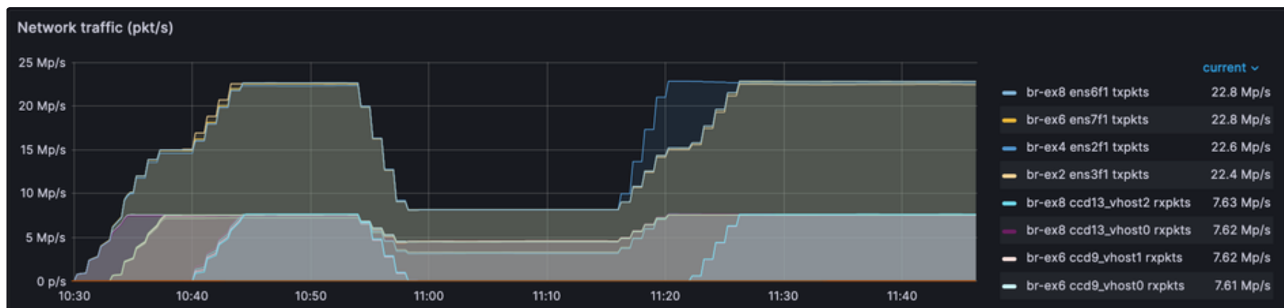


Figure 8 - Network Traffic MTU test (pkt/s)

The behavior of traffic measured in `pkt/s` is a direct inversion of traffic measured in `Gib/s`. This is shown in *Figure 8*, were the total amount of packages of MTU `64` being transported by each physical interfaces reaches `~23Mp/s`, with each individual virtual interface handling `~7.5Mp/s`. This values dip when increasing, to `~8Mp/s` and `~4Mps/s` respectively. After reverting the changes to packet size, the amount of packages transported increases again. Seeing this, combined with the amount of packet drops per interface, gives us a good

insight into the capabilities of OVS with DPDK.

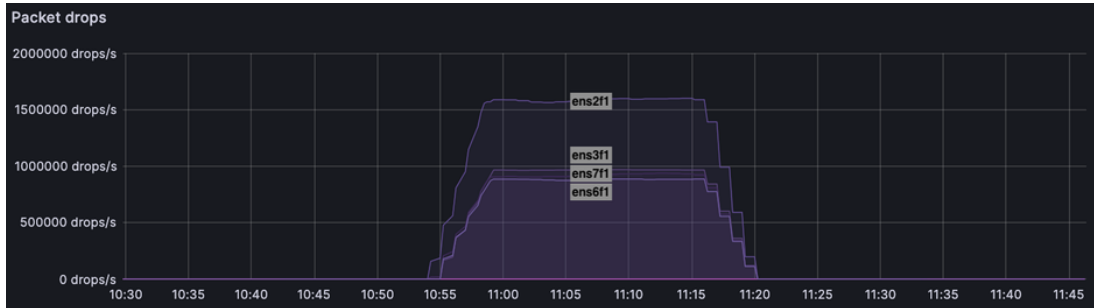


Figure 9 - Packet Drops MTU Test

From *Figure 9* we can observe that at the beginning of our second test, no drops of packages were occurring. Only when the MTU of the packages was increased to `1500` we started noticing drops, although, the drops were only present on physical interfaces. Drops then disappear when returning the packages to their original size. We can deduce from this fact that our DPDK dataplane is more than able to manage an amount of traffic larger than the network adapter's maximum. This can be seen as no drops occur in the virtual interfaces, meaning that DPDK is able to correctly transport the totality of packages being generated into the physical interfaces with no errors. Its only when the packages try to leave the server through the physical interfaces that we see drops, meaning that DPDK is able to manage more traffic than the physical interfaces are able to transport.

Considering that our dataplane is unnerved while handling heavy packages and knowing that it can also handle a much greater amount of individual packages, as per our MTU `64` test, it is sensible to hypothesize that if we installed more network adapters on the server we could reach an even greater volume of outbound traffic while using the same amount of vCPUs.

Finally, we should take a look at the CPU usage during our MTU test.

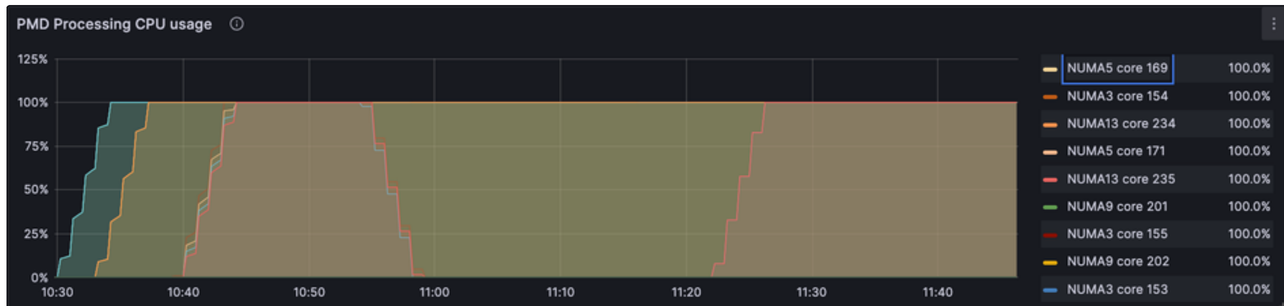


Figure 10 - PMD Processing CPU Usage

As exposed in *Figure 10*, the processor usage from cores where the DPDK vhost interfaces are pinned, immediately reaches `100%` when generating packages and stays the same throughout the experiment 's duration (except for vCPUs that were turned off in the MTU `1500` section of the test). We think this can be explained by two possible scenarios. First, all vCPUs handling traffic are currently at their limit during these tests, meaning that the installation of new network adapters could only increase the total amount of outbound traffic by the amount of drops being seen with MTU `1500` packages. Or a second, more optimistic, scenario might be that OVS is constantly checking the RX queue, creating the illusion that the processor is operating at maximum capacity, reaching `100%`; meaning that the amount of additional traffic we could achieve by using more network adapter might be even greater than expected!

Conclusion | Next Steps

As shown in this report, during the course of our tests we designed and configured an environment with the virtualization tools needed to launch virtual machines, generate packages on them and direct this traffic to an external network. This environment allowed us to test the traffic capabilities of OVS with a DPDK dataplane.

Our test showed that our architecture is able to handle a large volume of traffic originating from the VMs, transporting outbound packages efficiently and without drops, to the point of saturation of physical interfaces, which, by design, can not handle this amount of traffic and start dropping packages. Also, our tests with packages of MTU `64` indicate that OVS can manage a much greater amount of individual packages than what is observed with packages of MTU `1500`.

Since, currently, our main limit in the amount of traffic reaching the exterior of the server is the data rate of the installed network adapters, and no drops are being seen on virtual interfaces, the next logical step is to repeat this test using the same amount of vCPUs for generation and transportation of packages, but doing so while installing more network adapters. This would allow us to investigate the real limitations OVS has and how much traffic can be handled before we start seeing drops on virtual interfaces.

Following the same train of thought, if we are to repeat these experiments using more network adapters, it would also be natural to increase the amount of vCPUs in use too. Taking the configurations done in only four CCDs for this experiments, and replicating them for every CCD on the server would show us how much traffic we could generate and export from a single server. Based on the results presented in this report, and considering that this server has five slots for network adapters, if we were to fill all slots with `200Gb` dual port network adapters, we are confident that we should achieve a total amount of outbound traffic of approximately `920Gib/s`.

These are encouraging results, which showcase the deployed architecture's ability to handle sustained heavy traffic while providing enough new questions to inspire new research around the subject.