Contrail Performance Guide

Juniper Networks

- White paper -

Contrail Performance Guide

March 04th, 2020 - v3.2

v1	First version
v2	Updates: Page 14 (X710 NIC) and 52 (Problem with re assigning kernel driver back to interface when DPDK vRouter is stopped).
v2.1	Add description of vif command counters (July 2018)
v2.2	various changes: coremask for dpdk changed. vrouter logical cores numbering description.
v2.3	New appendix section with DPDK fine tuning for Intel NIC cards (Niantic and Fortville family).
v2.4	More details on DPDK
v2.5	Document rearrangement. Some updates on vrouter fine tuning options.
v2.6	Document rearrangement. More details about virtio.
v3.0	Performance tuning which will be available in 20.03 release.
v3.0.2	Added details of DPDK packet flow and polling core assignment as of R1910.
v3.0,3	Performance tuning parameters which will be available in 20.03 update
v3.1	Details about vNIC and MultiQueues - Control Thread new parameter (20.03 release).
3.1.1	Compute nodes and CPU capacity planning
3.2	Configurations description and deployment considerations sections rearrangement

Table of Contents

Contrail vrouter Dataplane Architecture 8

Performance requirements for packet processing 8

Contrail vrouter architecture 9

QEMU and Virtio 10

Compute node architecture 11

Non-Uniform Memory Access (NUMA) systems 11

Hyper-threading (HT) 12

Huge pages 13

Data Plane Development Kit (DPDK) 14

DPDK memory management 15

Contrail and DPDK 18

DPDK threads and Contrail Logical core numbering 23

DPDK and Poll Mode Drivers (PMD) 25

Linux drivers for PMD 26

UIO 26

VFIO 27

Contrail DPDK vrouter architecture 29

Contrail DPDK vRouter packet walk-through 29

DPDK VM incoming traffic (from underlay network) 31

Polling core - first step of packet processing 32

Forwarding core - second step of packet processing (MPLS over GRE) 33

Forwarding core - second step of packet processing (MPLS over UDP or VxLAN) 34

DPDK VM outgoing traffic 36

vNIC queues - Polling Core Assignment (R1910) 39

Packet Flow Summary (Based on R1910) 40

Supported NICs 42

Contrail related linux packages 43

Ubuntu 43

Redhat 43

Nova / Qemu / Libvirt 44

Configuration description 45

Compute node memory and huge pages 45

Huge pages configuration 45

Hugepage memory and NUMA sockets 47

Contrail vrouter and Single Hugepage size (Kubernetes) 49

Compute node processing and CPU allocation 50

Physical CPU cores allocation planning 50

Hyper Threading 50

CPU assignment and isolation mechanisms 51

isolcpus mechanism configuration 52

Tuned CPU partitioning configuration 53

DPDK vrouter process and threads 53

CPU assignment for DPDK vRouter 57

CPU assignment for DPDK vRouter packet processing threads 57

CPU assignment for DPDK vRouter service and control threads 59

contrail version <= 19.12 59

vRouter control and service threads pinning - contrail version >= 20.03 61

CPUs assigned to vrouter - Provisioning concerns 64

CPUs assigned to control and service threads (control and service cores) - 20.03 and later release 64

CPUs assigned to packet processing threads (forwarding cores) 66

CPU pinning and isolation 68

CPU isolation configuration (tuned and isolcpus) - RedHat 68

CPU isolation configuration (isolcpus) - Ubuntu 68

Operating system scheduler CPU assignment (CPUAffinity) - RedHat 69

Nova CPU assignment (vcpu_pin_set) 69

vRouter CPU assignment (CPU_LIST, SERVICE_CORE_MASK and DPDK_CTRL_THREAD_MASK) 70

Check CPU pinning on all processes 70

vNIC performance 71

CPU assigned for Multiqueue and virtIO 71

Single queue virtIO 71

Multiqueue virtIO 72

DPDK Multiqueue packet walk-through 72

vNIC queues queue size configuration (RedHat OpenStack) 73

multiQ vNIC and number of queues size configured (RedHat OpenStack) 74

Enable Multiqueue on OpenStack instances 75

How to change the number of queues for an OpenStack instances vNIC 76

Multiqueue and known limitations 77

Number of queues on VM instances: 77

Indirect descriptors are not supported: 79

IRQ setup 80

Irqbalance 80

Unsupported features 80

Deployment considerations 81

Define number of resources required for vRouter DPDK 81

Compute Node CPU capacity planning 81

Performance tests figures (DPDK vrouter) 81

vrouter forwarding cores capacity planning 85

vrouter DPDK control threads and services cores capacity planning 89

DPDK control threads 89

vRouter service threads 89

CPU assignment typical setups 90

Discover hardware architecture topology 92

Enable Jumbo 96

Host 96

Guest 96

Configure driver and network settings 96

Configure OpenStack 97

Configure nova scheduler 97

Define flavors for VNFs 97

Spawn a VM 98

Performance 99

Performance tuning essentials 99

Enable CPU performance mode 99

Manage a high flow environment 101

Increase flow table size 101

Flow setup rate optimisation 101

Configuration of packet based processing 103

vrouter physical bond setting 104

MPLSoUDP encapsulation Configuration 105

Contrail Configuration 105

SDN Gateway configuration: MPLSoUDP and Load Balancing 106

MPLSoUDP configuration 106

Virtual Network Load Balancing Configuration 108

DPDK buffer size adjustment (mbuf) 109

Intel recommendation to reduce or avoid packet loss 109

Problem with reassigning kernel driver back to interface when DPDK vRouter is stopped 110

Jumbo Frame support 111

Control the number of CPU cores vRouter 112

Appendixes 113

vrouter fine tuning parameters (kernel and DPDK mode) 113

Generic vrouter dimensioning parameters 113

32 bits vrouter nexthop limit parameter (contrail release 19.11 and later) 118

vrouter networking parameters (MTU) 119

vrouter DPDK fine tuning parameters 120

DPDK vrouter specific parameters 120

DPDK fine tuning rules (mainly for Intel NIC) 123

DPDK vouter parameters provisioning 126

DPDK vouter parameters configuration onto the compute node (5.0 and later versions) 127

Connectivity Troubleshooting 130

How to check vrouter connectivity 130

vRouter vif queues setup 132

vRouter vif queues activation 134

Packet drop troubleshooting 136

Interface traffic counters 136

Packet Drop statistics 143

vif command: get-drop-stats option 149

Dropstats and packet drop log utility 150

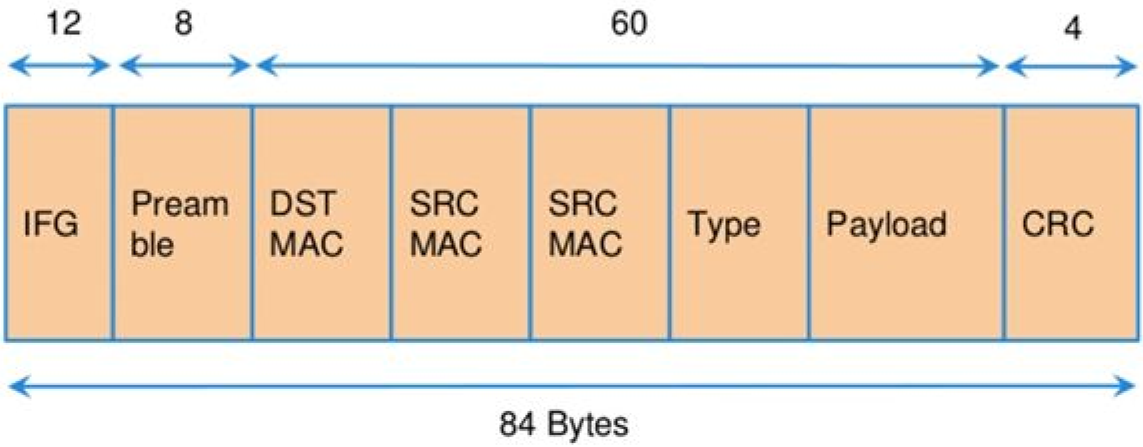
Faulty vrouter deployment troubleshooting 153

References 154

Contrail vrouter Dataplane Architecture

Performance requirements for packet processing

Ethernet minimum frame size is 64 Bytes. When Ethernet frames are sent onto the wire, Inter Frame Gap and Preamble bits are added. Minimum size of Ethernet frames on the physical layer is 84 Bytes (672 bits).



For a 10 Gbit/s interface, the number of frames per seconds can reach up to 14.88 Mpps for traffic using the smallest Ethernet frame size. It means a new frame will have to be forwarded each 67 ns.

A CPU running at 2Ghz has a 0.5 ns cycle. Such a CPU has a budget of only 134 cycles per packet to be able to process a flow of 10 Gb/s.

Generic Linux Ethernet drivers are not performant enough to be able to process such a 10Gb/s packet flow.

Indeed lots of times are required to :

- perform packet processing in Linux Kernel using interrupt mechanism,
- transfer application data from host memory to Network Interface card

A SDN solution like Contrail needs to use specific setup and mechanisms in order to be able to process network flows onto a generic x86 platform at a high rate.

Contrail vrouter architecture

vRouter is made of 2 parts :

- **vRouter agent** : vrouter control and configuration plane
- **vRouter dataplane** : vrouter userplane (user packet processing)

vRouter agent is used to manage the communication between Contrail SDN Controller and vRouter.

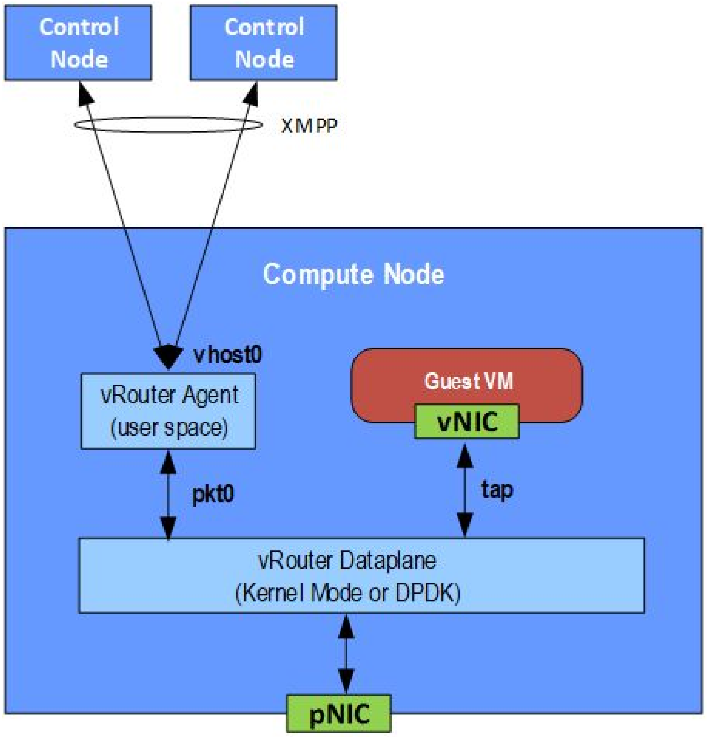
vRouter agent has 2 interfaces :

- vhost0 (north controller)
- pkt0 (south dataplane)

vRouter agent is always running in compute node user mode.

vRouter dataplane has 2 kinds of interfaces

- Physical NIC : connected to underlay network in order to send traffic out of the compute node
- vNIC tap : connected to guest virtual instances



vRouter dataplane is running either in Linux Kernel space or in Linux User space when DPDK is used. Better performances are expected when vRouter dataplane is running into Linux User space.

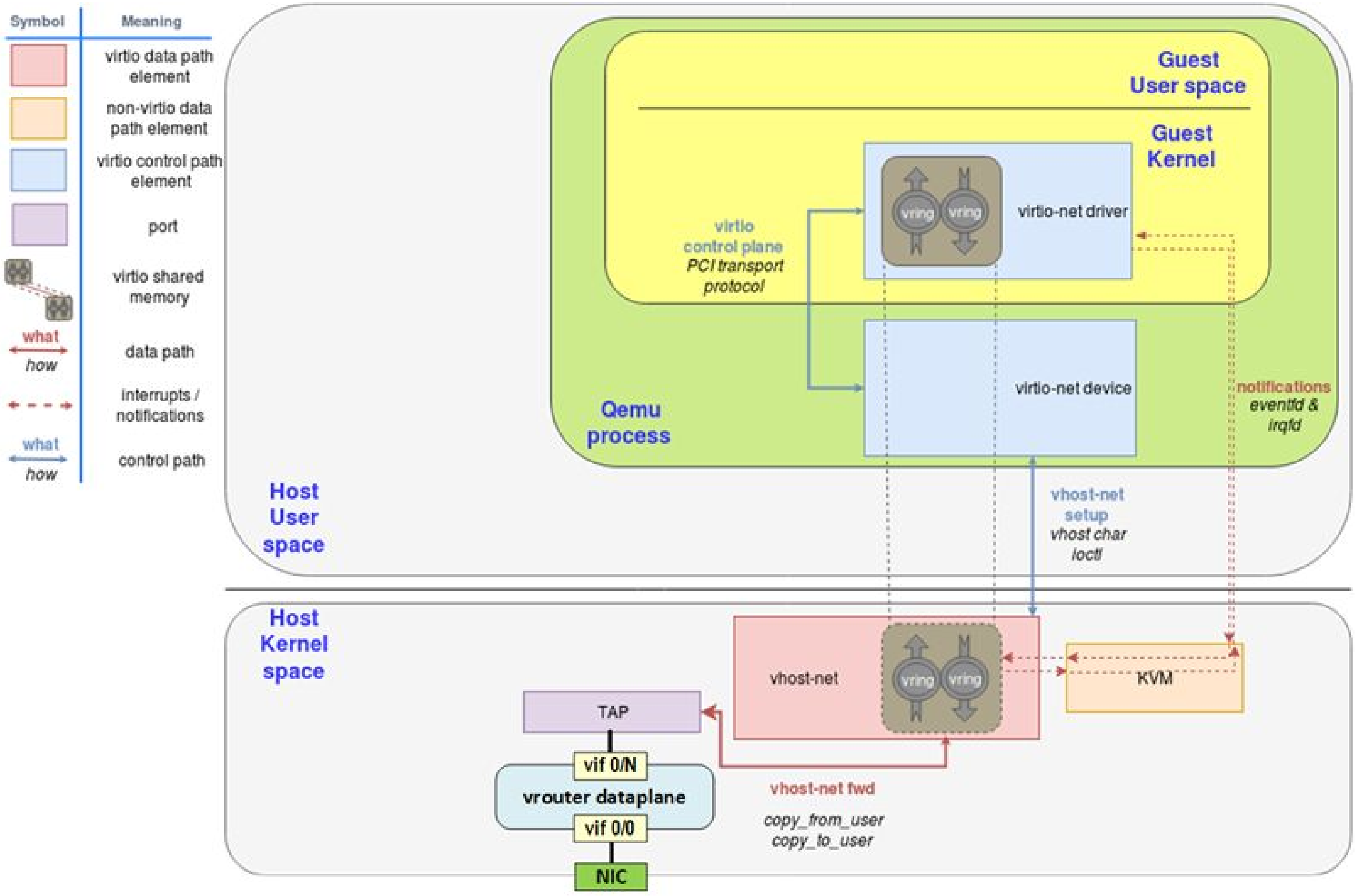
QEMU and Virtio

Virtio was developed as a standardized open interface for virtual machines (VMs) to access simplified devices such as block devices and network adaptors.

There are two parts to networking within VirtIO:

- the virtual network device that is provided to the guest (e.g. a PCI network card).
- the network backend that interacts with the emulated NIC (e.g. puts packets onto the host's network, to the vrouter).

In Contrail Networking, VirtIO is used to connect with guest VM vNIC onto the vrouter vif interface. In the diagram below you have a detailed view of a Virtual instance connectivity with a Kernel mode vrouter :



For more information on VirtIO :
https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net
https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net

Compute node architecture

Non-Uniform Memory Access (NUMA) systems

A traditional server has a single CPU, a single RAM and a single RAM controller.

A RAM can be made of several DIMM banks in several sockets, all being associated to the CPU. When the CPU needs access to data in RAM, it requests it to its RAM controller.

Recent servers can have multiple CPUs, each one having its own RAM and its own RAM controller. Such systems are called NUMA systems, or Non-Uniform Memory Access. For example, in a server with 2 CPUs, each one can be a separate NUMA: NUMA0 and NUMA1.

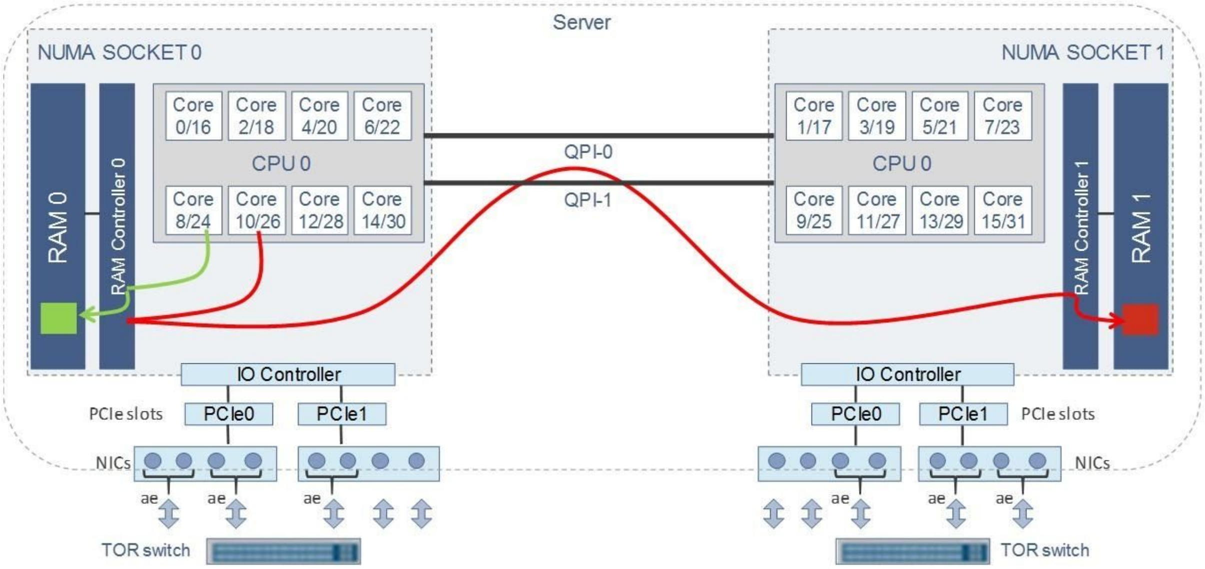


Figure. NUMA nodes architecture.

In green: CPU core accessing a memory item located in its own NUMA's RAM controller, showing minimum latency.

In red: CPU core accessing a memory item located in the other NUMA through the QPI (Quick Path Interconnect) path and the remote RAM controller, showing a higher latency.

When CPU0 needs to access data located in RAM0, it will go through its local RAM controller 0. Same thing happens for CPU1. When CPU0 needs to access data located in the other RAM1, the first (local) controller 0 has to go through the second (or remote) RAM controller 1 which will access the (remote) data in RAM 1. Data will use an internal connection between the 2 CPUs called QPI, or Quick Path Interconnect, which is typically of a high enough capacity to avoid being a bottleneck, typically 1 or 2 times 25GBps (400 Gbps). For example the Intel Xeon E5 has 2 CPUs with 2 QPI links between them; Intel Xeon E7 has 4 CPUs, with a single QPI between pairs of CPUs.

The fastest RAM that the CPU has access to is the register, which is inside the CPU and reserved to it. Beyond the register, the CPU has access to cached memory, which is a special memory based on higher performance hardware. Cached memory are shared between the cores of a single CPU. Typical characteristics of memory cache are:

- Accessing a Level 1 cache takes 7 CPU cycles (with a size of 64KB or 128KB).
- Accessing a Level 2 cache takes 11 CPU cycles (with a size of 1MB).
- Accessing a Level 3 cache takes 30 CPU cycles (with a larger size).

If the CPU needs to access data that is in the main RAM, it has to use its RAM controller. Access to RAM takes then typically 170 CPU cycles (the green line in the diagram). Access to the remote RAM through the remote RAM controller typically adds 200 cycles (the red line in the diagram), meaning RAM latency is roughly doubled. When data needed by the CPU is located both in the local and in the remote RAM with no particular structure, latency to access data can be unpredictable and unstable.

Hyper-threading (HT)

A single physical CPU core with hyper-threading appears as two logical CPUs to an operating system. While the operating system sees two CPUs for each core, the actual CPU hardware only has a single set of execution resources for each core. Hyper-threading allows the two logical CPU cores to share physical execution resources. The sharing of resources allows two logical processors to work with each other more efficiently, and allows a logical processor to borrow resources from a stalled logical core (assuming both logical cores are associated with the same physical core). Hyper-threading can help speed processing up, but it’s nowhere near as good as having actual additional cores. The performance of vRouter with ***sibling*** HT cores can increase by 10% to 20% (result is based on performance tests described hereinafter).

Huge pages

Memory is managed in blocks known as pages. On most systems, a page is 4Ki. 1Mi of memory is equal to 256 pages; 1Gi of memory is 256,000 pages, etc. CPUs have a built-in memory management unit that manages a list of these pages in hardware.

The Translation Lookaside Buffer (TLB) is a small hardware cache of virtual-to-physical page mappings. If the virtual address passed in a hardware instruction can be found in the TLB, the mapping can be determined quickly.

If not, a TLB miss occurs, and the system falls back to slower, software based address translation. This results in performance issues. Since the size of the TLB is fixed, the only way to reduce the chance of a TLB miss is to increase the page size.

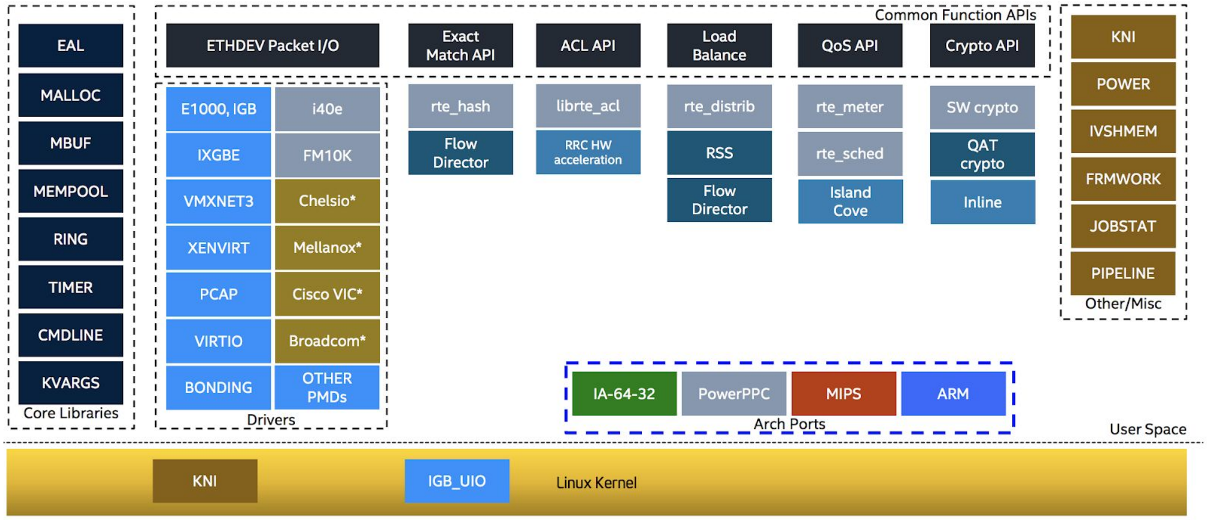
Virtual memory address lookup slows down when the number of entries increases.

A huge page is a memory page that is larger than 4Ki. In x86_64 architecture, in addition to **standard 4KB memory** page size, two larger page sizes are available: **2MB** and **1GB**.

Contrail DPDK vrouter can use both or only one huge page size.

Data Plane Development Kit (DPDK)

Data Plane Development Kit (DPDK) is a set of data plane libraries and network interface controller drivers for fast packet processing, currently managed as an open-source project under the Linux Foundation.



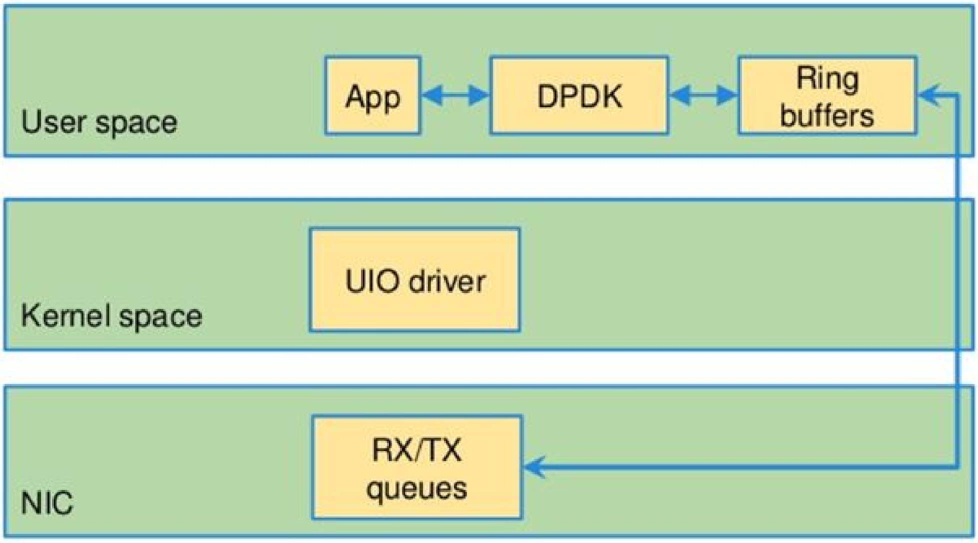
The main goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications. The framework creates a set of libraries for specific environments through the creation of an Environment Abstraction Layer (EAL), which may be specific to a mode of the Intel® architecture (32-bit or 64-bit), Linux* user space compilers or a specific platform. These environments are created through the use of make files and configuration files. Once the EAL library is created, the user may link with the library to create their own applications.

The DPDK implements a run to completion model for packet processing, where all resources must be allocated prior to calling Data Plane applications, running as execution units on logical processing cores. The model does not support a scheduler and all devices are accessed by polling. The primary reason for not using interrupts is the performance overhead imposed by interrupt processing.

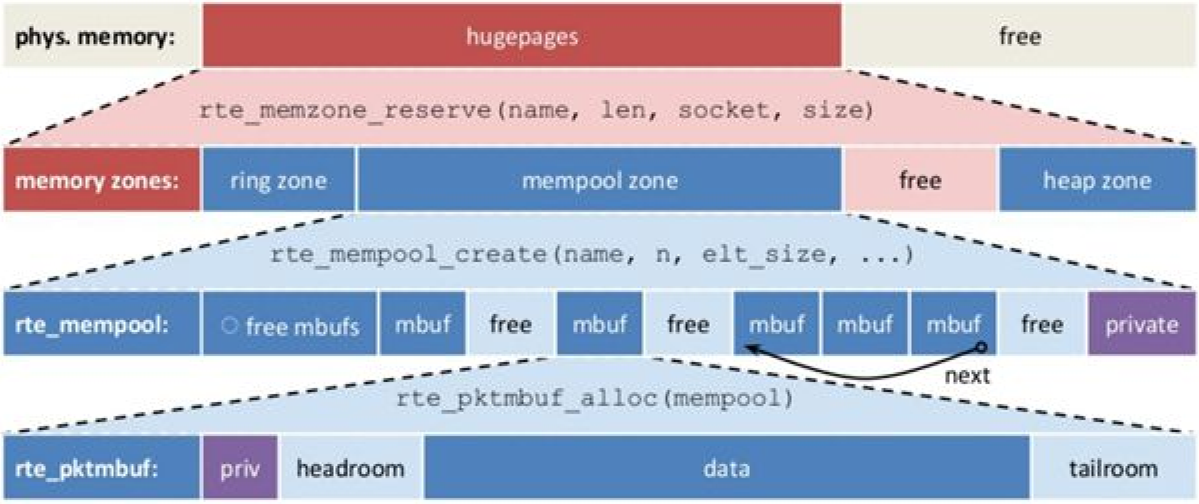
For more information please refer to dpdk.org documents http://dpdk.org/doc/guides/prog_guide/index.html.

DPDK memory management

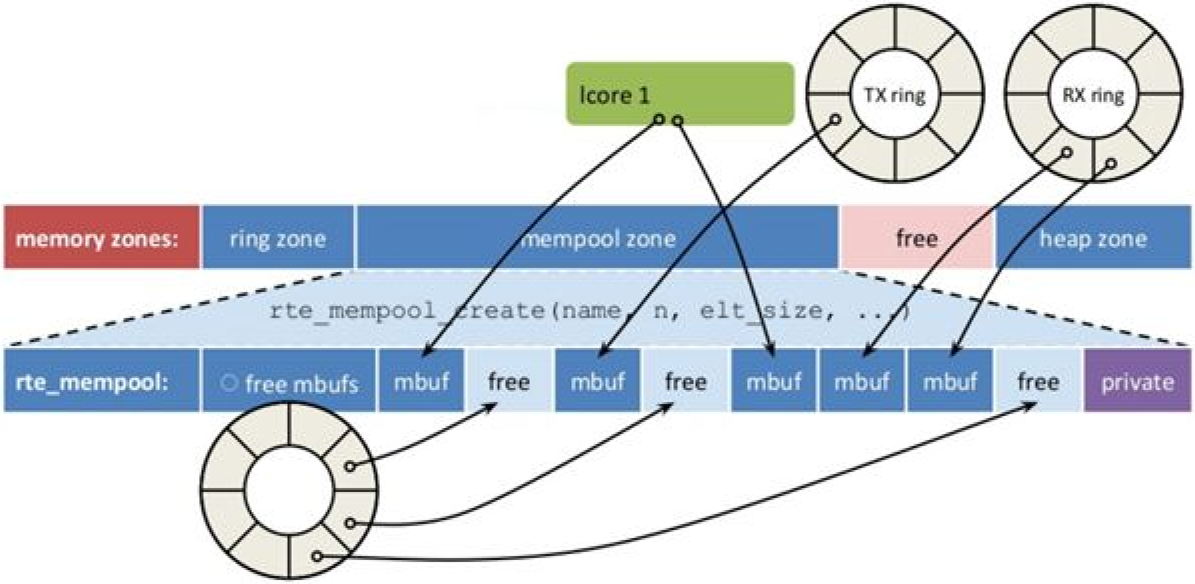
With DPDK there is a direct link between application data stored in host memory and the NIC memory used to transfer data onto the wire :



DPDK uses message buffers known as mbufs to store packet data into the host memory. These mbufs are stored in memory pools known as mempools. Mempools are set up as a ring, which creates a pool with a configuration similar to a first-in, first-out (FIFO) system.



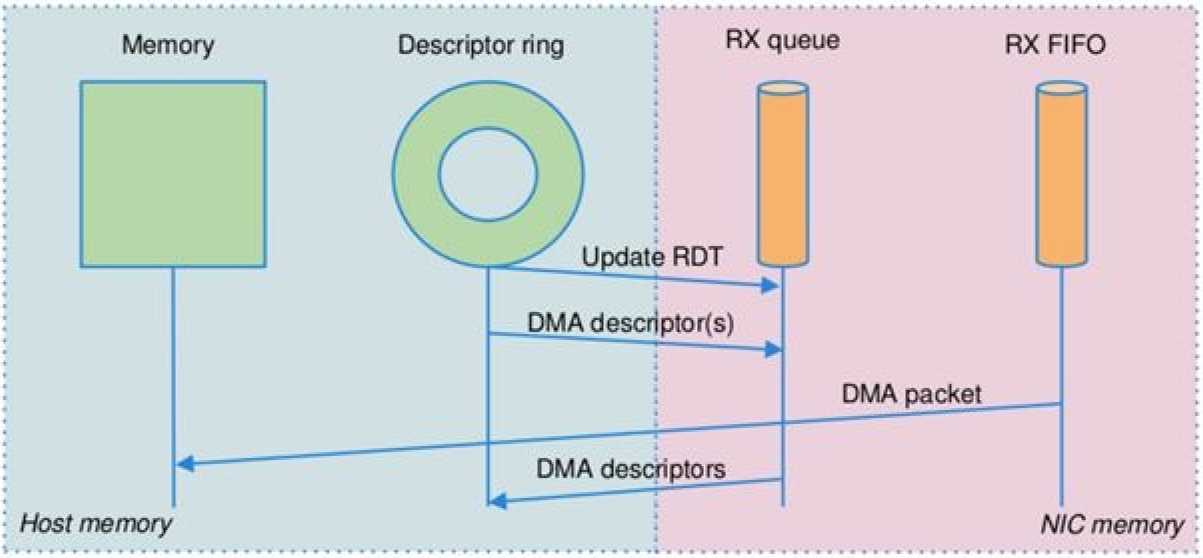
Rings descriptors are managing data storage into mempools. The more descriptors RX/TX rings are containing, the more memory size will be required in each mempool to store data.



The Host OS exchanges packets with the NIC through the so called rings. A ring is a circular array of descriptors allocated by the OS in the system memory (RAM). Each descriptor contains information about a packet that has been received or that is going to be transmitted.

RX ring

RX ring is managing Data transfer from NIC memory to host memory :

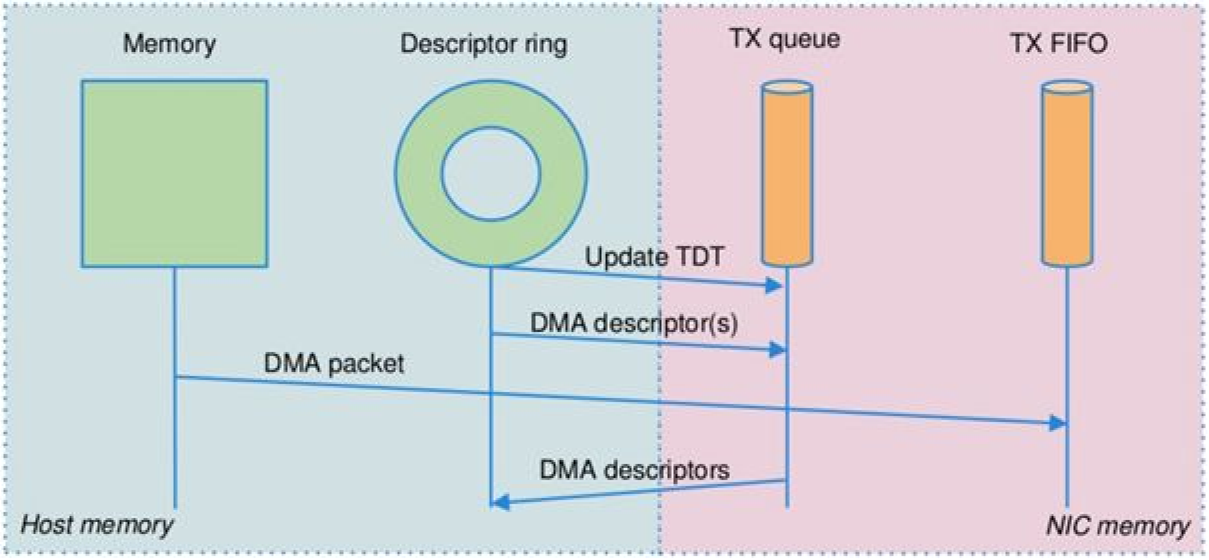


Synchronization between the OS and the NIC happens through two registers, whose content is interpreted as an index in the RX ring:

- Receive Descriptor Head (RDH): indicates the first descriptor prepared by the OS that can be used by the NIC to store the next incoming packet.
- Receive Descriptor Tail (RDT): indicates the position to stop reception, i.e. the first descriptor that is not ready to be used by the NIC.

TX ring

TX ring is managing data transfer from host memory to NIC memory :



Synchronization between the host OS and the NIC happens through two registers, whose content is interpreted as an index in the TX ring:

- Transmit Descriptor Head (TDH): indicates the first descriptor that has been prepared by the OS and has to be transmitted on the wire.
- Transmit Descriptor Tail (TDT): indicates the position to stop transmission, i.e. the first descriptor that is not ready to be transmitted, and that will be the next to be prepared.

DMA

Direct Memory Access (DMA allows PCI devices to read (write) data from (to) memory without CPU intervention. This is a fundamental requirement for high performance devices.

Contrail and DPDK

Contrail vRouter is using DPDK library to improve packet processing performance. Starting from Contrail 3.2.5 DPDK library 17.02 is used. Earlier Contrail versions were based on DPDK library version 2.1 (https://www.juniper.net/documentation/en_US/contrail3.2/information-products/topic-collections/release-notes/jd0e36.html#jd0e185).

Contrail DPDK library fork
https://github.com/Juniper/contrail-dpdk

DPDK release notes
http://dpdk.org/doc/guides-17.02/rel_notes/index.html

When using contrail vrouter without DPDK, vRouter dataplane process is running into Linux Kernel.

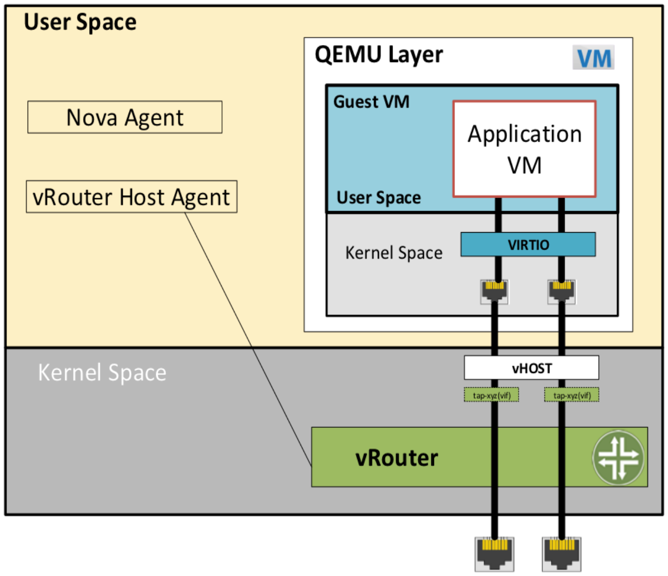


Figure. vRouter in kernel with VM.

When using contrail vrouter without DPDK, vRouter dataplane is running into Linux Kernel.

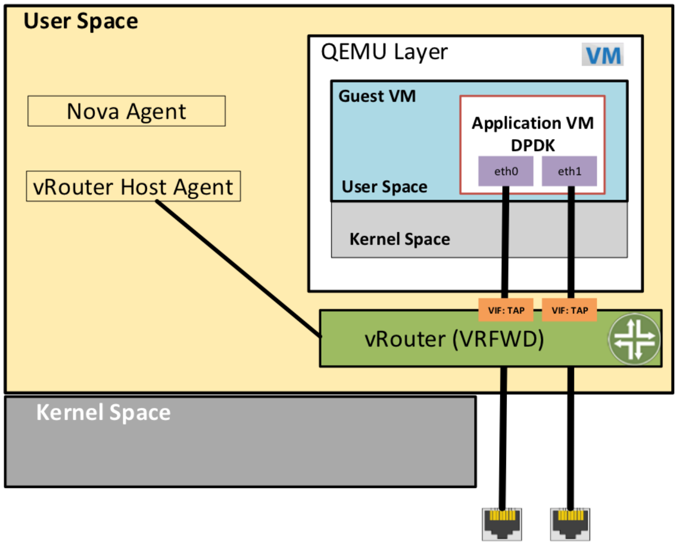


Figure. vRouter DPDK with VM application DPDK aware.

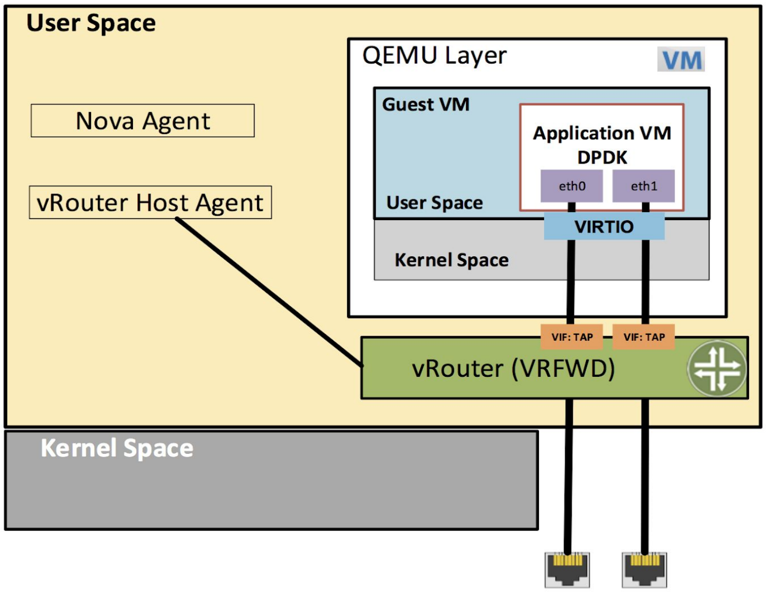
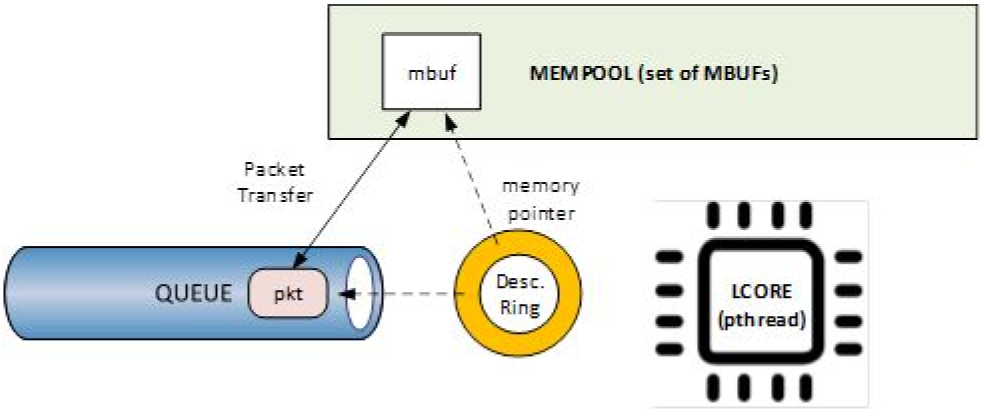


Figure. vRouter DPDK with VM application not DPDK aware.

Contrail DPDK vrouter as any DPDK application is based on queue management. Queue packets processing consists in :

- queues onto a network interface
- pthreads (logical core) for packet enqueue or dequeue (packet polling)
- descriptor rings (TX or RX) for packet transfer from interface queue to host memory (mbuf) or from host memory (mbuf) to interface queue
- memory spaces (mempool)



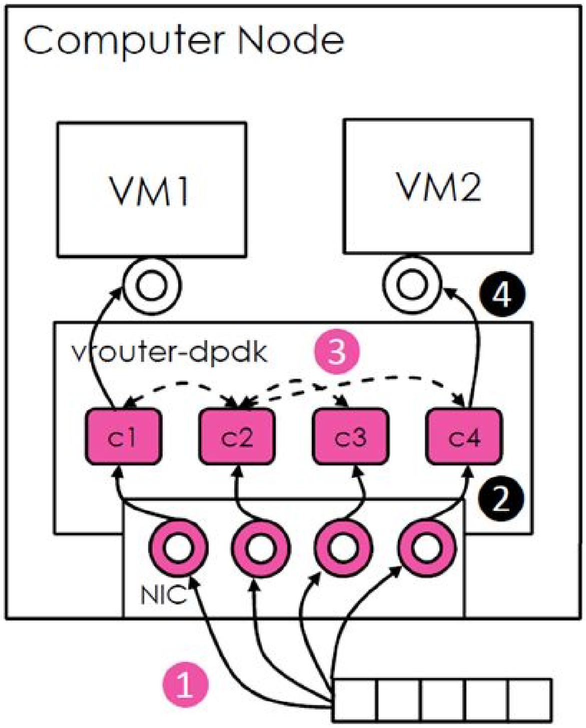
For each vrouter interface one or several queues are managed. 3 kinds of interfaces are connected onto the vrouter :

- *Physical interface card* (usually a bond) : vif 0/0
- *Internal processing interfaces* (pkt0) : vif 0/2
- *Virtual Machine Interfaces* : vif 0/n

Two kinds of interfaces have to be taken in consideration :

- *vrouter physical interface card* : one TX queue and one RX queue are created for each CPU pinned to vrouter. Onto vrouter virtual machine.
- *vrouter virtual machine interface* : one or several queues could be implemented for each virtual NIC. It depends if the given virtual machine is supporting multi-queue or not.

In the diagram below we have a first overview of vrouter packet processing for an external packet to be sent to virtual machine instances.



There are 4 main steps in packet processing :

- **step one**: incoming packets are put into RX queues by the network interface card.
- **step two:** each CPU (c1 to c4) is polling its allocated queue in order to put each packet into host memory in order to be processed.
- **step three:** packet processing is achieved by a forwarding thread. This forwarding thread could be performed onto any CPU allocated to vrouter.
- **step four:** after being processed, packet is copied onto virtual machine TX interface queues.

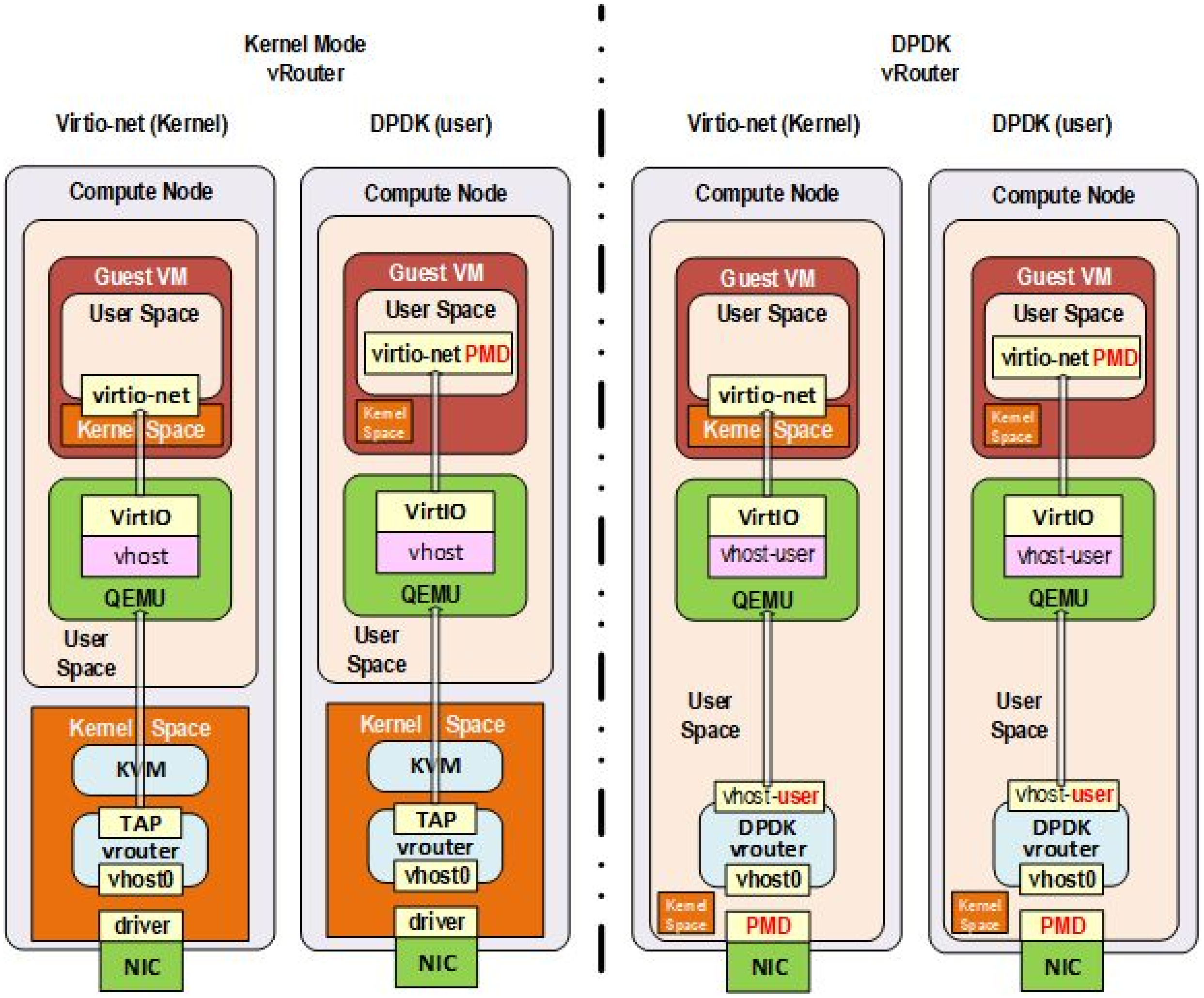
PS: these 4 steps are performed for VxLAN and MPLSoUDP. For MPLSoGRE, since RSS (hashing algorithm) computed by the NIC is giving a unique answer for most of the flows there is no traffic spreading across CPU (step 2). In this particular case there is only one polling CPU. A hash is calculated onto the incoming decapsulated packets (inner packet) in order to spread them on several “processing cores” (step 3).

Virtual instances and DPDK (VirtIO)

In order to get good network performances, virtual instances (especially VNF - Virtual Network Function) are implementing DPDK. 4 topologies are possible :

- vrouter in Kernel mode collecting VNF with a network in Kernel Mode
- vrouter in Kernel mode collecting VNF running DPDK
- vrouter in DPDK mode collecting VNF with a network in Kernel Mode
- vrouter in DPDK mode collecting VNF running DPDK

An all DPDK stack, used both at vrouter level and at VNF level, is providing the best performance.



When DPDK is used into the virtual instance, vNIC driver (virtio-net) used for packet processing in Kernel space is replaced by a virtio Poll Mode Driver (virtio-net PMD) in order to perform packet processing in Virtual Instance user space.

DPDK threads and Contrail Logical core numbering

DPDK vrouter is a multi-threads application.

There are 3 kinds of threads into a DPDK vrouter :

- **control threads** : used for DPDK internal processing.
  eal-intr-thread, rte_mp_handle, rte_mp_async
- **service threads** : used for connectivity between vrouter agent and vrouter forwarding plane (DPDK vrouter). Thread names are lcore 0 to 9
- **processing threads** : used for packet polling and processing (forwarding plane)
  thread names are lcore 10 and above

The term “lcore” refers to an EAL (Environment Abstraction Layer) thread, which is really a Linux/FreeBSD pthread (physical Thread). A numbering into the dpdk vrouter is used for lcores.

This ***lcore numbering*** used in vrouter can be seen into source file (vr_dpdk.h).

An enumeration is defining this numbering :

```
enum {
    VR_DPDK_KNITAP_LCORE_ID = 0,
    VR_DPDK_TIMER_LCORE_ID,
    VR_DPDK_UVHOST_LCORE_ID,
    VR_DPDK_IO_LCORE_ID,        = 3
    VR_DPDK_IO_LCORE_ID2,
     VR_DPDK_IO_LCORE_ID3,
    VR_DPDK_IO_LCORE_ID4,
    VR_DPDK_LAST_IO_LCORE_ID,   # 7
    VR_DPDK_PACKET_LCORE_ID,    # 8
    VR_DPDK_NETLINK_LCORE_ID,
    VR_DPDK_FWD_LCORE_ID,       # 10
};
```

PS: Lots of other details concerning DPDK vrouter can also be seen into following files :
include/vr_dpdk.h
dpdk/dpdk_vrouter.c
dpdk/vr_dpdk_lcore.c

The 3 first logical numbers (0 to 2) are used for “service lcores”. The 5 next ones (3 to 7) are booked for IO lcores. The 2 next ones (8 to 9) are “lcores with TX queues”. Logical cores with number 10 and above are forwarding logical cores.

0 to 9 lcore numbers are statically defined into the source code and can’t be configured by user.
lcore numbers 10 and above are used for forwarding purpose and defined with CPU affinity value. This value is set into *contrail-vrouter-dpdk.ini* configuration file.

```
command=/bin/taskset <CPU Affinity> /usr/bin/contrail-vrouter-dpdk  ….
```

For instance 0x000154000154 CPU affinity is giving following mapping :

```
32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
 1  0  1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  1  0  1  0  1  0  0
```

Host CPU numbers 2, 4, 6, 8, 26, 28, 30 and 32 are used for vrouter forwarding logical cores.

When the router boots up, it displays a message about its logical cores numbering. For instance, we can get into *contrail-vrouter-dpdk-stdout.log* file, such a message :

```
VROUTER:      --lcores  "(0-2)@(0-47),(8-9)@(0-47),10@2,11@4,12@6,13@8,14@26,15@28,16@30,17@32"
EAL: Detected 48 lcore(s)
VROUTER: Using 8 forwarding lcore(s)
VROUTER: Using 0 IO lcore(s)
VROUTER: Using 5 service lcores
```

First part of the message - (0-2)@(0-N),(8-9)@(0-N) - is always the same, at this internal CPU logical numbering is statically defined in vrouter source code. N is the total number of CPU available in the compute onto which the vrouter is running.

Last part of the message - **10@2,11@4,12@6,13@8,14@26,15@28,16@30,17@32**- is depending on the CPU affinity. Here 8 logical cores are used for forwarding purpose, they are numbered from 10 to 17. They are mapped one by one onto the host CPUs 2, 4, 6, 8, 26, 28, 30 and 32.

PS: This is this vrouter logical core numbering that has to be in dropstats command (not the real host CPU numbering)

```
$ dropstats –core 10
```

PS: A maximum of 16 polling cores is currently supported by Contrail as the maximum RX queue is currently 16:

```
#define VR_DPDK_MAX_NB_RX_QUEUES    16
```

DPDK and Poll Mode Drivers (PMD)

When DPDK is used, Network interfaces are no more managed in Kernel space. Legacy NIC driver which is usually used to manage the NIC has to be replaced by a new driver which is able to run into user space. This new drive, called Poll Mode Driver (PMD) will be used to manage the network interface into user space with the DPDK library.

A Poll Mode Driver consists of APIs, provided through the BSD driver running in user space, to configure the devices and their respective queues. In addition, a PMD accesses the RX and TX descriptors directly without any interrupts (with the exception of Link Status Change interrupts) to quickly receive, process and deliver packets in the user's application.

Some PMD are being used to manage physical interfaces :

I40e PMD for Intel X710/XL710/X722 10/40 Gbps family of adapters
http://dpdk.org/doc/guides/nics/i40e.html

IXGBE PMD
http://dpdk.org/doc/guides/nics/ixgbe.html

Linux bonding PMD
http://dpdk.org/doc/guides/prog_guide/link_bonding_poll_mode_drv_lib.html

Some PMD are being used to manage virtual interfaces :

Virtio PMD
http://dpdk.org/doc/guides/nics/virtio.html

Linux drivers for PMD

In Linux user space environment, the DPDK application runs as a user-space application using the pthread library. PCI information about devices and address space is discovered through the */sys kernel* interface and through kernel modules such as uio_pci_generic, igb_uio or vfio-pci.

Different PMDs may require different kernel drivers in order to work properly. Depending on the PMD being used, a corresponding kernel driver should be loaded and bound to the network ports. Before loading, make sure that each NIC has been flashed with the latest version of NVM/firmware.

UIO

Supported NICs

- Intel igb (82575, 82576, 82580, I210, I211, I350, I354, DH89xx)
- Intel ixgbe (82598, 82599, X520, X540, X550)
- Intel i40e (X710, XL710, X722)

*Note: RHEL does not support "**uio_pci_generic**" driver*

To enable igb_uio driver change physical_uio_driver in /etc/contrail/contrail-vrouter-agent.conf file and restart supervisor-vrouter.

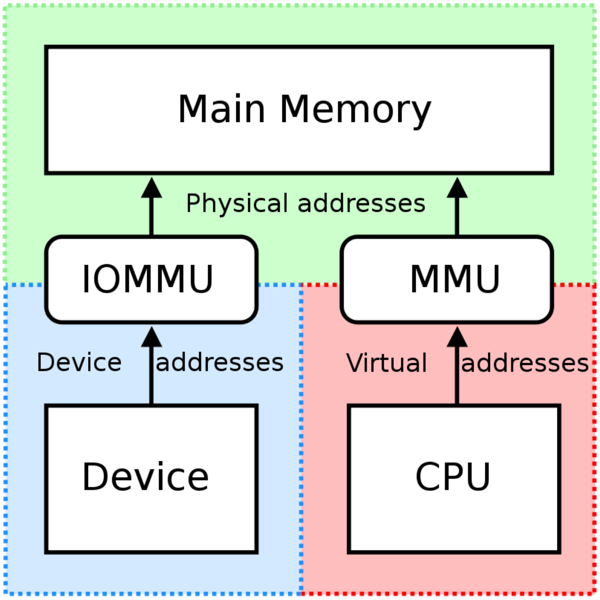
```
[DEFAULT]
physical_uio_driver=igb_uio
```

VFIO

Supported NICs

- Intel i40e (X710, XL710, X722)

IOMMU (Input–Output Memory Management Unit (IOMMU) is a memory management unit (MMU) that connects a Direct Memory Access (DMA) capable I/O bus to the main memory.
In Virtualization, an IOMMU is re-mapping the addresses accessed by the hardware into a similar translation table that is used to map guest-physical address to host-physical addresses.



IOMMU provides a short path for the guest to get access to the physical device memory.Intel has published a specification for IOMMU technology as Virtualization Technology for Directed I/O, abbreviated VT-d.

VFIO need to get IOMMU enabled :

- both kernel and BIOS must support and be configured to use IO virtualization (such as Intel® VT-d).
- IOMMU must be enabled into Linux Kernel parameters in /etc/default/grub and run update-grub command.

GRUB configuration example :

```
GRUB_CMDLINE_LINUX_DEFAULT="iommu=pt intel_iommu=on"
```

VFIO can be also be used without IOMMU While this is just as unsafe as using UIO, it does make it possible for the user to keep the degree of device access and programming that VFIO has, in situations where IOMMU is not available.

To enable vfio-pci driver change physical_uio_driver in /etc/contrail/contrail-vrouter-agent.conf file and restart supervisor-vrouter.

```
[DEFAULT]
physical_uio_driver=vfio-pci
```

Drivers features compatibility list

	RHEL DPDK	**Ubuntu DPDK**	**RHEL SRIOV (VF)***	**Ubuntu SRIOV (VF)***
igb_uio	No (no dkms support)	Yes (dkms)	No	Yes
uio_pci_generic	No (not supported by RHEL)	Yes	No	No
vfio_pci	Yes	Yes	Yes	Yes

*vRouter in parallel with SRIOV (VF support on VM)

Contrail DPDK vrouter architecture

Contrail DPDK vRouter packet walk-through

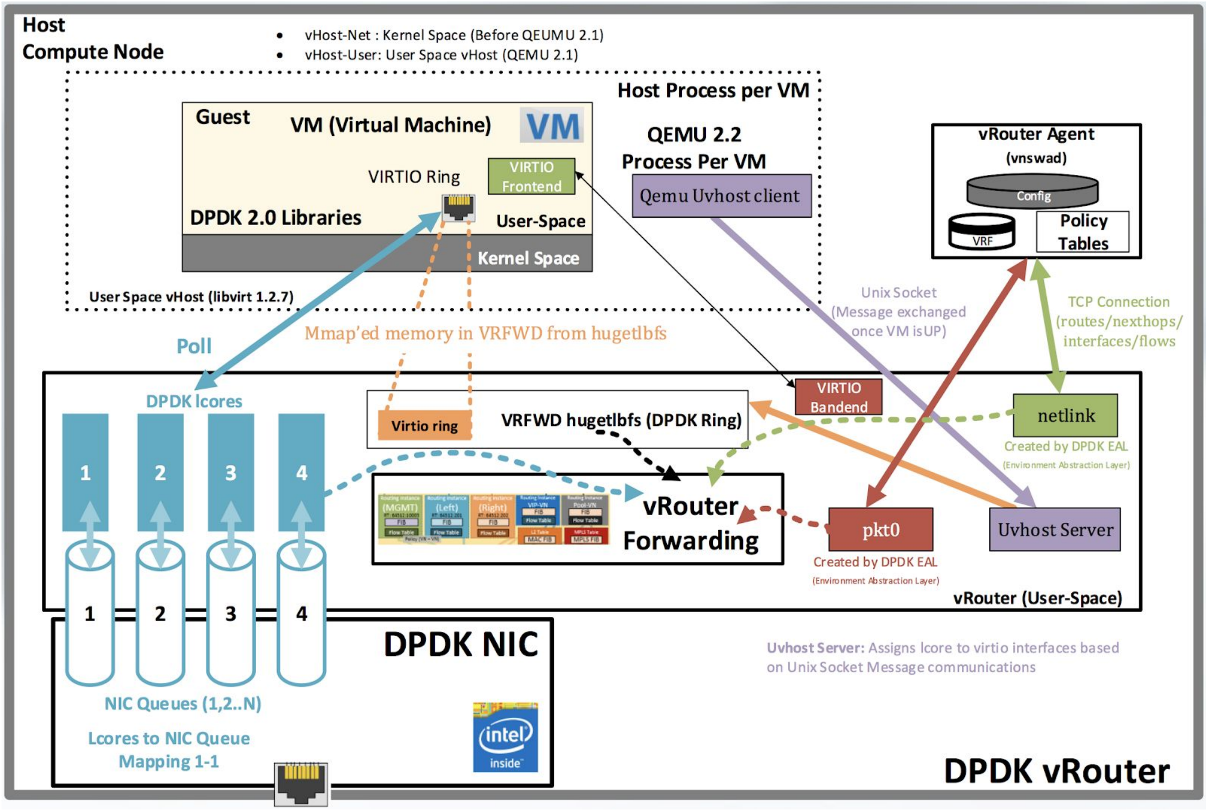


Figure. Contrail DPDK vRouter architecture.

Contrail DPDK vRouter runs forwarding threads which are used to poll NIC queues (1 to 1 mapping between thread and queue). Each forwarding thread is pinned to its dedicated CPU core (named DPDK *lcore*). The number of NIC queues is dependent on the number of CPU cores which are specified by coremask in Contrail DPDK vRouter configuration (*contrail-vrouter-dpdk.ini file*). Contrail DPDK vRouter uses DPDK ethdev function to program the NIC.

Forwarding threads in their infinity loops poll its queues on the NIC to check if there is packet or burst of packets to receive. It is providing descriptors where packets have to be copied to host memory (memory allocated by hugepage). That packet movement to memory is done using NIC DMA (Direct Memory Access processor). Forwarding threads are lcore ID 10 and above (cf appendix section for lcore/pthread numbering)

Besides forwarding threads Contrail DPDK vRouter runs service threads :

- **pkt0 thread** for a communication with Contrail vRouter Agent (flow setup)
  this is vRouter lcore ID 8 thread (cf appendix section for lcore/pthread numbering). This is used to send and receive packets to/from the agent. Eg: 1st packet of flow, arp, dhcp etc.

- **netlink thread** to give vRouter Agent possibility to get statistics from Contrail DPDK vRouter
  this is vRouter lcore ID 9 thread (cf appendix section for lcore/pthread numbering)
  The Netlink socket family is a Linux kernel interface used for inter-process communication (IPC)

Before contrail 20.03 release, Service threads were pinned to the whole range of cores available on the host system. Since the 20.03 release it is possible to pin these threads on some specific CPUs.

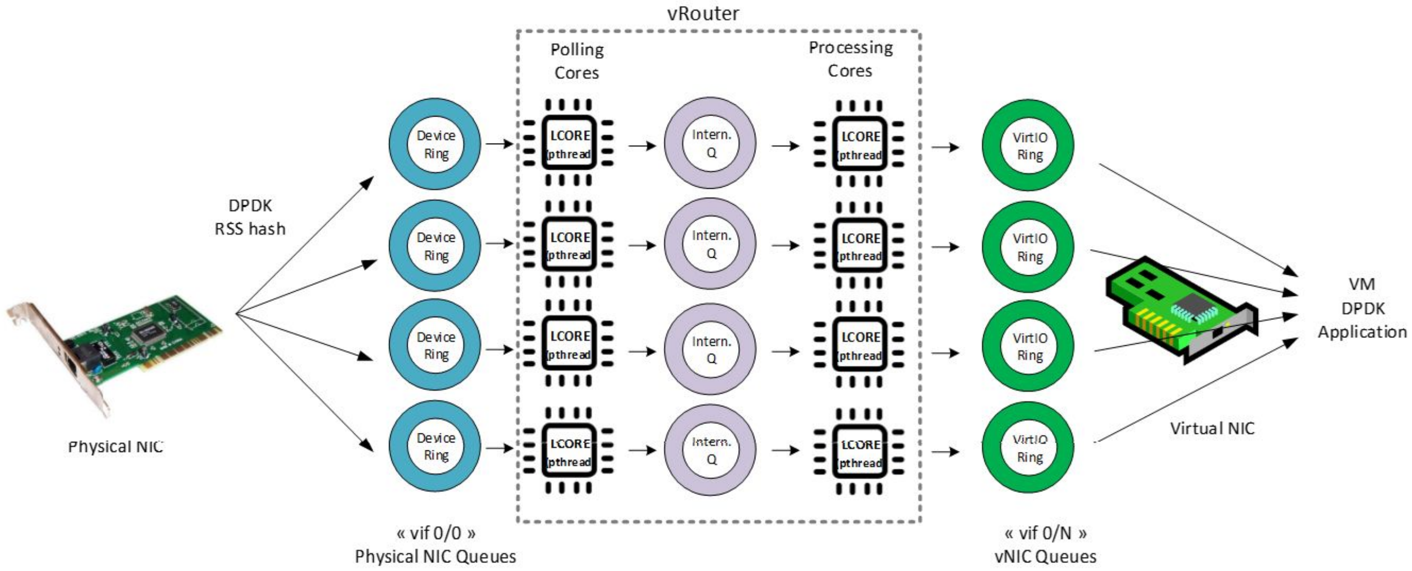
Physical NIC (usually an interface bond) itself spread packets across its queues using 5-tuple hashing function (*source IP, destination IP, source port, destination port and protocol*). Depending on overlay encapsulation protocol used, the incoming traffic is well balanced or not onto NIC card RX queues :

- **MPLSoUDP, VxLAN encapsulation protocols** : are providing a good entropy. A good hashing will be achieved with their UDP datagrams (several source port values are used for the same tunnel IP source and destination pair).
- **MPLSoGRE encapsulation protocol** : is not providing a good entropy. Packets from one SDN gateway will be placed only in a single queue, which is not an optimal host resource utilization. With a second gateway it is likely to have two NIC queues utilized.

That is the reason Juniper recommends implementing MPLSoUDP on Juniper MXes (supported in Junos >=16.2).

DPDK VM incoming traffic (from underlay network)

When the traffic is coming from the underlay network, encapsulated packets are received DPDK compute Physical interface which is bound to vrouter vif 0/0:



Incoming packets are processed in 3 steps :

- packets are polled by a vrouter polling core and decapsulated.
- decapsulated packet are sent by polling core to a processing core
- processing core is delivering packet to a destination vif interface

Two main situations have to be considered for processing core selection :

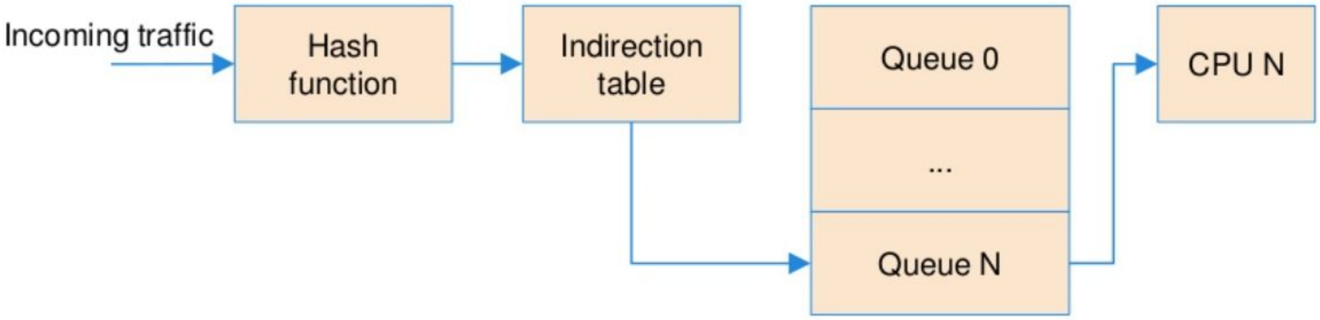
- incoming underlay packets are encapsulated with MPLS over GRE
- incoming underlay packets are encapsulated with MPLS over UDP or VxLAN

Polling core - first step of packet processing

The packet arrives at a physical NIC (pNIC) encapsulated in a MPLSoGRE header (outer header).The pNIC computes a hash on the packet outer header

- this hash is called “RSS hash” (Receive Side Scaling)
- this hash is computed onto the encapsulated packet and does not take into account the header of the tenant packet (inner header)

Based on the “RSS hash”, the pNIC selects a queue and writes the packet to the queue (RX QUEUE into the previous diagram).



Then incoming packet is dequeued by the polling logical core (step 1) :

- there is one queue per vRouter logical core
- vRouter polling logical core in charge of that queue manages packet transfer from NIC queue to a free mbuf into the mempool.
- when incoming packets are encapsulated with MPLS over GRE, vRouter polling logical core in charge of that queue reads the packet header, computes a hash, and selects a vRouter processing logical core based on the hash. Hashing algorithm is in vr_dpdk_ethdev.c module (https://github.com/Juniper/contrail-vrouter/blob/R5.0/dpdk/vr_dpdk_ethdev.c#L873)

Then the forwarding logical core is processing the packet (step 2). Forwarding CPU is in charge to do :

- flow lookup,
- flow enforcement (switching, routing, NAT, packet transformation)
- decapsulation,
- packet delivery to VM RX queue (vRouter vif TX-QUEUE)

Forwarding core - second step of packet processing (MPLS over GRE)

In a first stage polling CPU bound to the queue on which a packet has been received, is triggering packet transfer from given NIC RX queue to mbuf. This CPU (here CPU 0) will also choose which CPU (forwarding CPU) will have to process the packet (decapsulation, routing, switching, …).
So a hash will be calculated onto polling CPU on decapsulated packet in order to select a forwarding CPU (decapsulated packet should have a better entropy than encapsulated one with MPLS over GRE).

In the diagram below it is shown a situation where the polling CPU core is selecting another CPU for packet processing. This would be the case for MPLSoGRE traffic, since the hash performed by the NIC is likely not efficient :

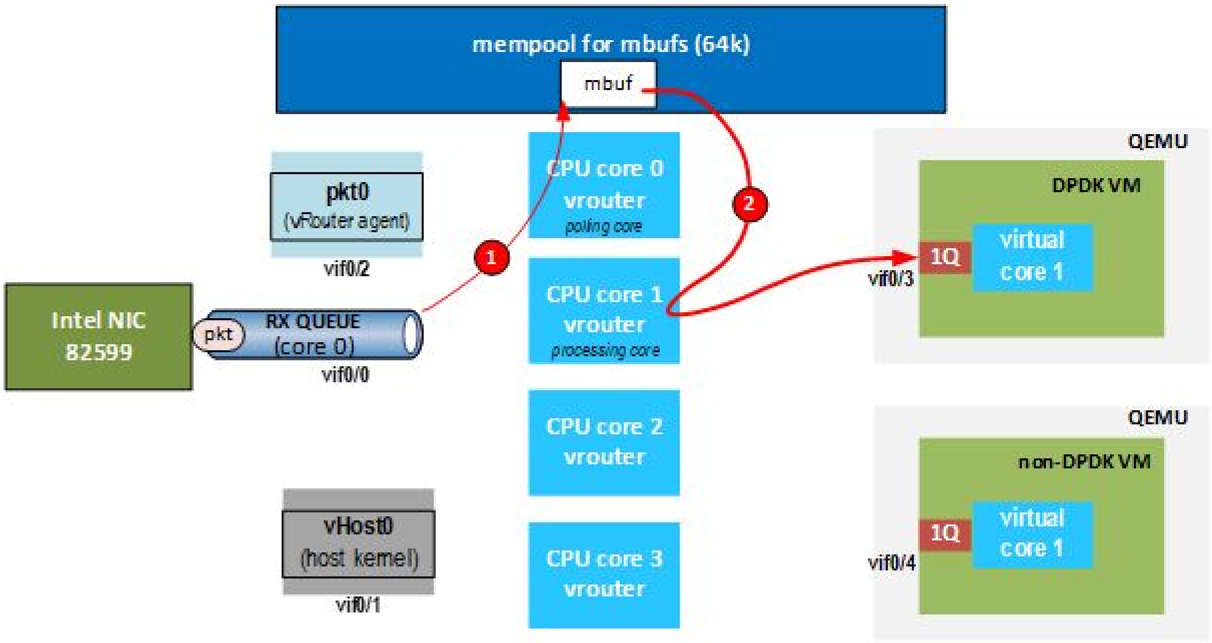


Figure. DPDK Contrail vRouter packet walk-through from NIC to DPDK guest.

Forwarding core - second step of packet processing (MPLS over UDP or VxLAN)

In a first stage polling CPU bound to the queue on which a packet has been received, is triggering packet transfer from given NIC RX queue to mbuf. This CPU (here CPU 0) will also choose which CPU (forwarding CPU) will have to process the packet (decapsulation, routing, switching, …).

But as incoming encapsulated packets are using UDP protocol, a good entropy is expected for incoming encapsulated traffic. No hash is recalculated on the poling core, and each incoming packet will be processed on the same forwarding core than the polling one.

In the diagram below, it is shown a situation where the polling CPU core is the same as the forwarding CPU, which will be the case for MPLSoUDP and VXLAN packets from the NIC:

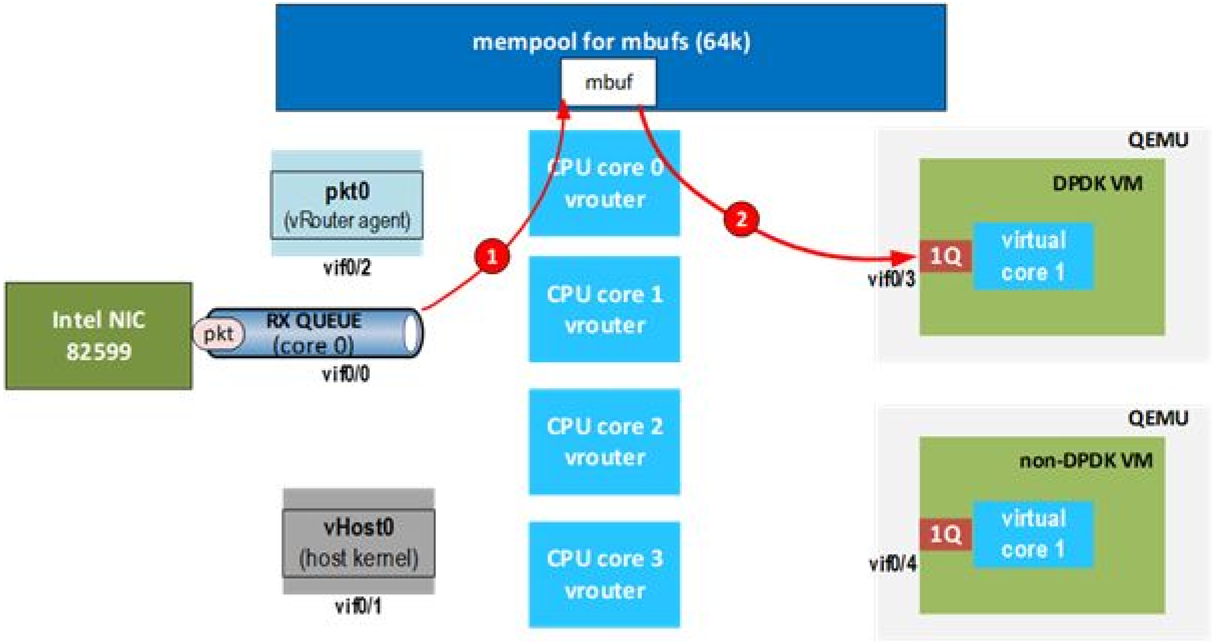
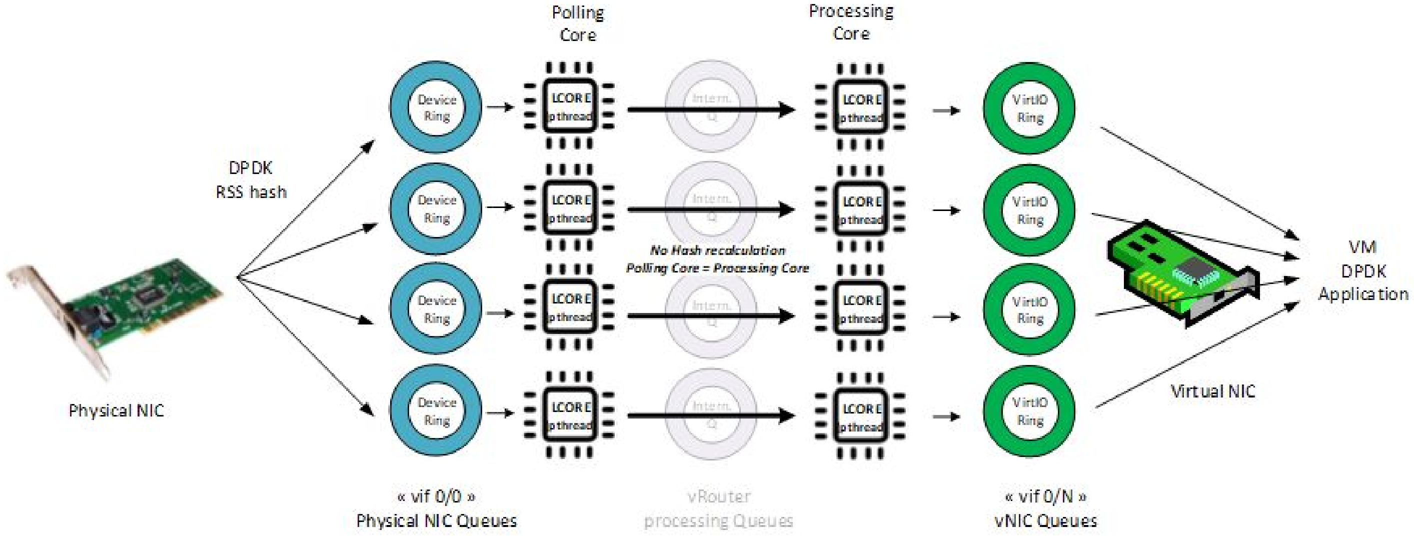
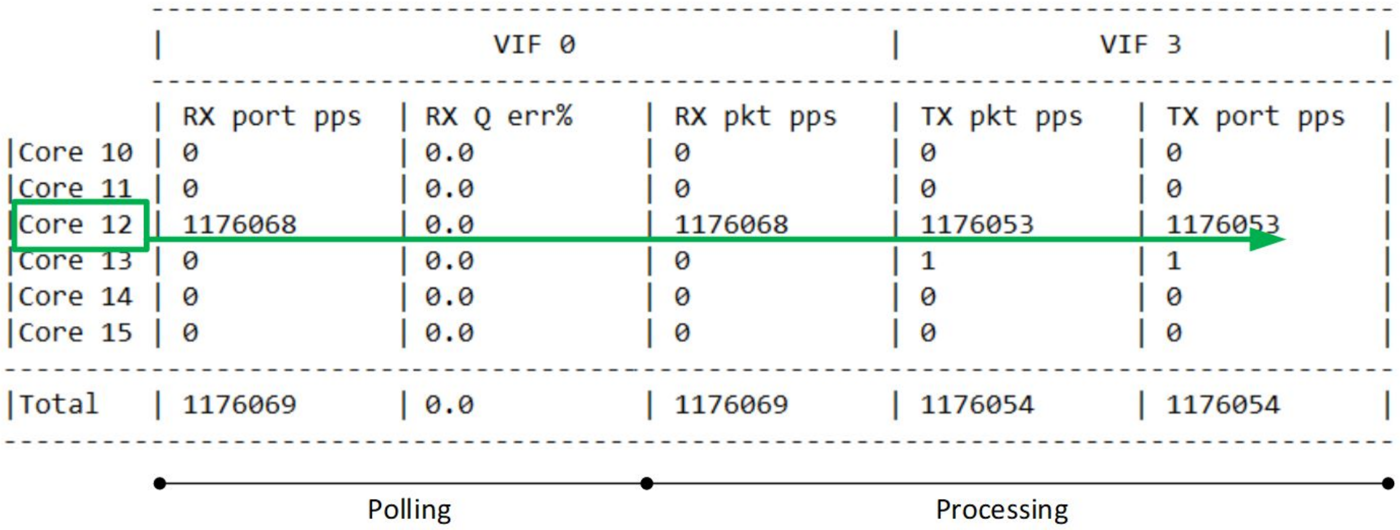


Figure. DPDK Contrail vRouter packet walk-through from NIC to DPDK guest.

When MPLS over UDP is used, there is internal traffic load balancing onto vrouter CPU. Incoming packets are processed with the same forwarding core as polling core :



It can be easily shown using a traffic generator to send a "single UDP" flow onto a virtual instance. When single UDP flow is reaching the vrouter, RX port packets and RX packets counters on vif 0/0 have the same values for a same lcore: Here the VM incoming traffic is polled and processed by core 12:



```
          --------------------------------------------------------------------------
          |                  VIF 0                    |             VIF 3          |
          --------------------------------------------------------------------------
          | RX port pps   | RX Q err%    | RX pkt pps   | TX pkt pps   | TX port pps   |
|Core 10  | 0             | 0.0          | 0            | 0            | 0             |
|Core 11  | 0             | 0.0          | 0            | 0            | 0             |
|Core 12  | 1176068       | 0.0          | 1176068      | 1176053      | 1176053       |
|Core 13  | 0             | 0.0          | 0            | 1            | 1             |
|Core 14  | 0             | 0.0          | 0            | 0            | 0             |
|Core 15  | 0             | 0.0          | 0            | 0            | 0             |
------------------------------------------------------------------------------------
|Total    | 1176069       | 0.0          | 1176069      | 1176054      | 1176054       |
------------------------------------------------------------------------------------
```

Table above has been made using real single flow traffic seen on each vif 0/0 and vif 0/3 interface (vNIC) with *vif --get* command (cf appendixes)

DPDK VM outgoing traffic

vRouter forwarding threads are also responsible for polling virtio interface queues. The decision which thread will poll which virtio queue is made by Contrail DPDK vRouter. If VM uses multiqueue then more vRouter forwarding threads (CPU cores) will be engaged in receiving packets from a single VM.

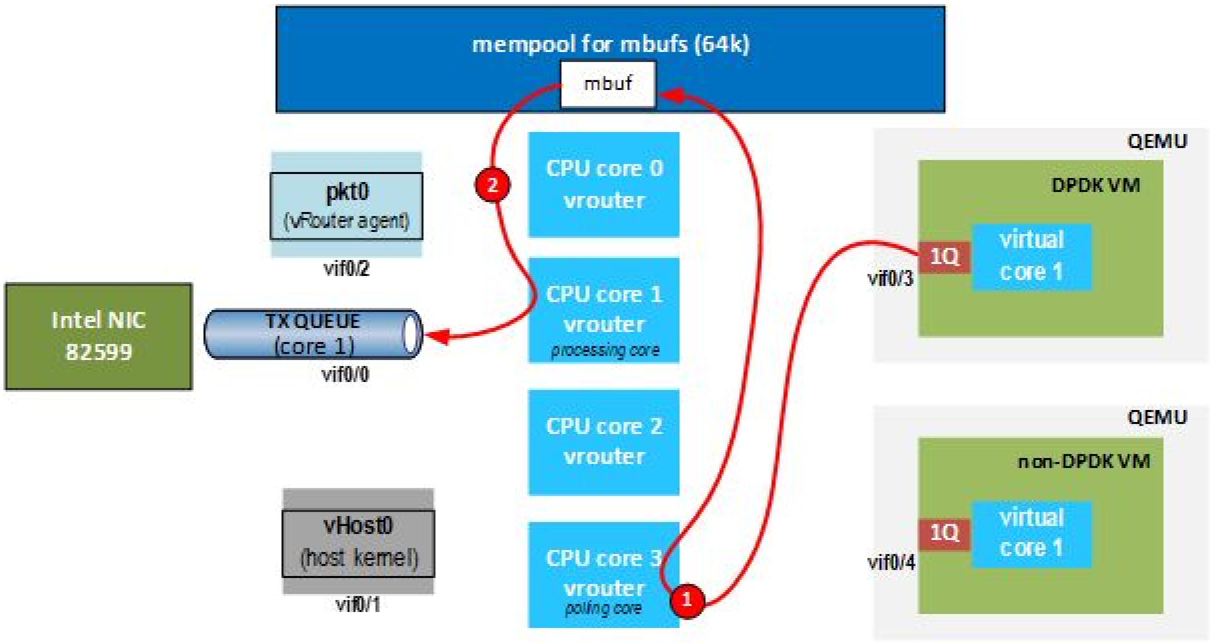


Figure. DPDK Contrail vRouter packet walk-through from DPDK guest to NIC.

When polling CPU core (in the above example CPU core 3) polls the packet from VM virtio interface queue it selects which forwarding CPU core will process the incoming packet. This forwarding CPU core does flow lookup, flow enforcement, encapsulation and packet delivery to NIC transmit queue.

The packet is placed on a vNIC TX queue (vRouter vif RX queue) by the Virtual machine. A given vNIC TX queue is always polled by the same vRouter logical core. A round robin algorithm (described in next section) is used to assign vNIC queues to logical cores. vNIC sub-interfaces are sharing TX/RX queues with their parent interface. Hence the same logical core is polling these queues.

Then, the incoming packet is dequeued by the polling logical core (step 1) :

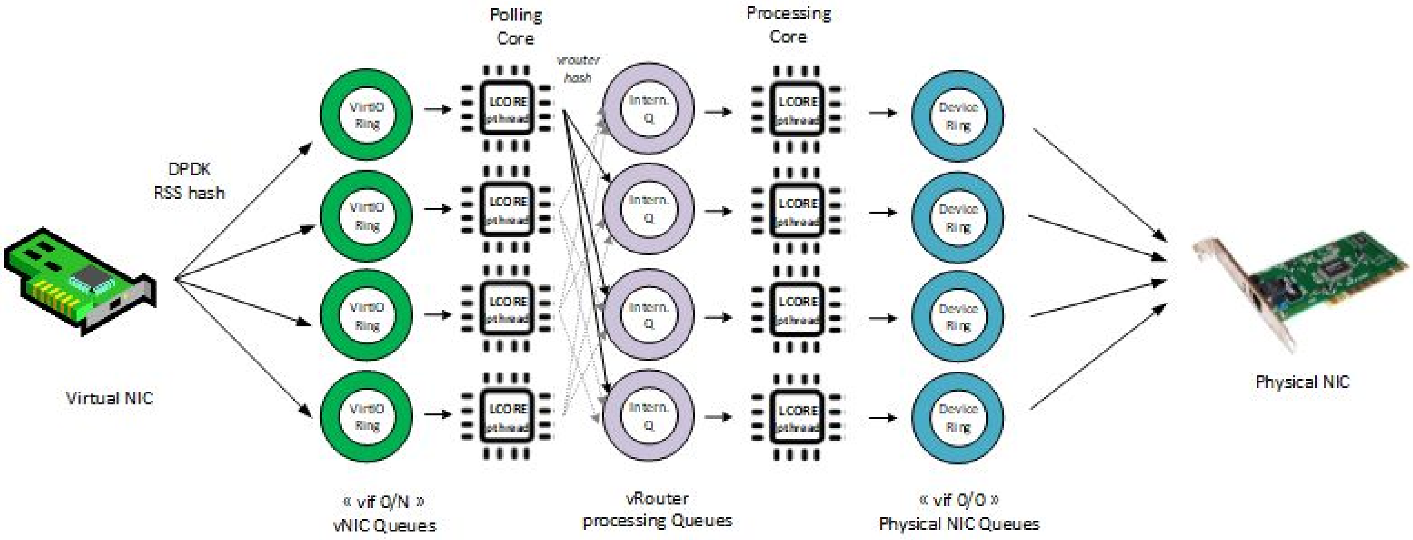
- vRouter polling logical core in charge of that queue reads the packet header,
- computes a hash, and selects a vRouter processing logical core based on the hash. Hashing algorithm is in vr_dpdk_ethdev.c module (https://github.com/Juniper/contrail-vrouter/blob/R5.0/dpdk/vr_dpdk_ethdev.c#L873) RSS hashing depends on the packet type:
    - for non-UDP/TCP IP packets (except GRE), a 2-tuple is used for the hash: source IP address, destination IP address

    - for GRE, a hash is done using the source IP address and destination IP address, followed by a hash for the GRE key if present
    - for TCP or UDP IP packets (IPv4 or IPv6), a hash is done for the source IP address and destination IP address followed by a hash for the source port and destination port
    - Ethernet information is not taken into consideration for hash computation:
        - VLAN tag (eg if a sub-interface is used), are not used for the hash
        - source and destination MAC addresses are not used for the hash
    - Non-IPv4/IPv6 packets (i.e. L2 packets) will not have a hash performed and will be processed by their polling core.

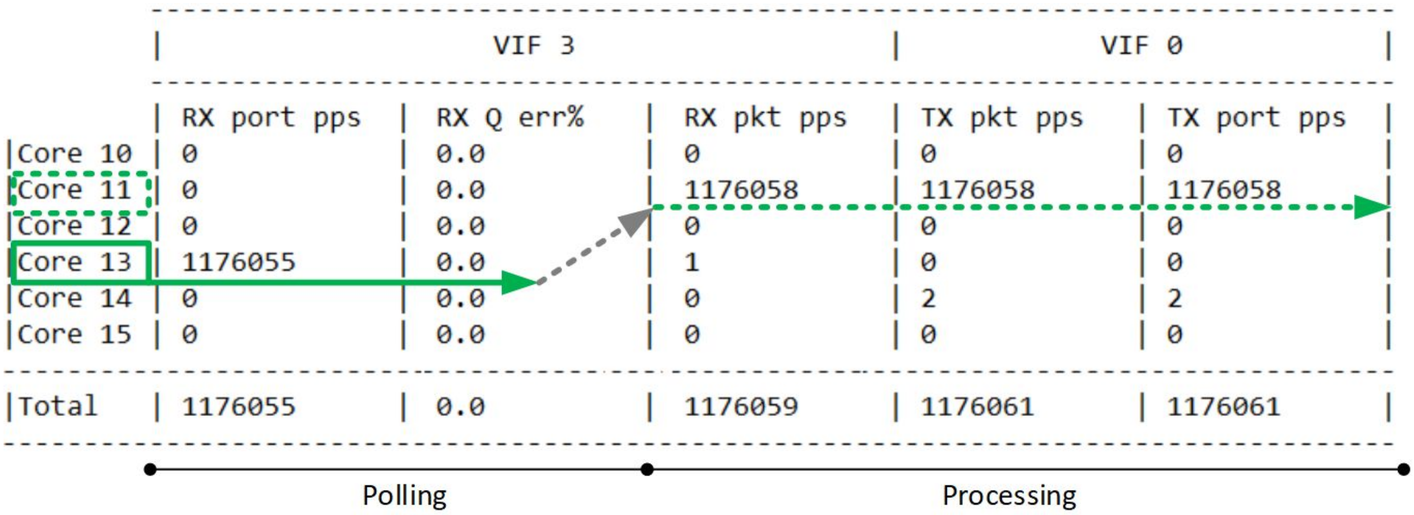
Then forwarding logical core is processing the packet (step 2). Forwarding CPU is in charge to do :

- flow lookup,
- flow enforcement (switching, routing, NAT, packet transformation)
- encapsulation,
- packet delivery to physical NIC TX queue

For VM incoming traffic a hash is calculated in order to rebalance received traffic on a given polling core to all other forwarding cores :



It can be easily shown using a traffic generator to send a “single UDP” flow onto a virtual instance. When single UDP flow is reaching the vrouter, RX port packets and RX packets counters on vif 0/N have the same values for distinct lcores. Here the VM incoming traffic is polled by core 13 and processed by core 11:



```
        -------------------------------------------------------------
        |                  VIF 3                |         VIF 0        |
        -------------------------------------------------------------
        | RX port pps | RX Q err% | RX pkt pps | TX pkt pps | TX port pps |
|Core 10 | 0          | 0.0       | 0          | 0          | 0           |
|Core 11 | 0          | 0.0       | 1176058    | 1176058    | 1176058     |
|Core 12 | 0          | 0.0       | 0          | 0          | 0           |
|Core 13 | 1176055    | 0.0       | 1          | 0          | 0           |
|Core 14 | 0          | 0.0       | 0          | 2          | 2           |
|Core 15 | 0          | 0.0       | 0          | 0          | 0           |
--------------------------------------------------------------------
|Total   | 1176055    | 0.0       | 1176059    | 1176061    | 1176061     |
--------------------------------------------------------------------
```

Table above has been made using real single flow traffic seen on each vif 0/0 and vif 0/3 interface (vNIC) with *vif --get* command (cf appendixes)

vNIC queues - Polling Core Assignment (R1910)

vNIC queues are assigned to logical cores in the following way:

- The forwarding core that is currently polling the least number of queues is selected, with a tie won by the core with the lowest number (the first forwarding core is lcore 10)
- A queue is created for each forwarding core, starting with the least used core and wrapping around to the start of the forwarding cores after the maximum is reached. However, only the first queue is actually enabled. All of the other queues will only be used if the VM supports multiqueue and enables them.
- If the VM supports multiqueue, then it enables the additional queues, at which point they are mapped to the forwarding core they were assigned when they were created.

Log messages showing this process can be seen in the /var/log/containers/contrail/contrail-vrouter-dpdk.log file.

When interface is added and the queue to forwarding core mappings are made. There are six forwarding cores in this setup. Only the first queue is actually enabled at this point. The remaining queues will use the mentioned forwarding cores if the VM chooses to enable them (i.e. it supports multiqueue):

```
2020-01-22 13:53:28,385 VROUTER:        lcore 11 RX from HW queue 0
2020-01-22 13:53:28,385 VROUTER:        lcore 12 RX from HW queue 1
2020-01-22 13:53:28,385 VROUTER:        lcore 13 RX from HW queue 2
2020-01-22 13:53:28,385 VROUTER:        lcore 14 RX from HW queue 3
2020-01-22 13:53:28,385 VROUTER:        lcore 15 RX from HW queue 4
2020-01-22 13:53:28,385 VROUTER:        lcore 10 RX from HW queue 5
```

When a VM (in this case a vSRX) requests to enable some of its available queues. The vrings here correspond to both transmit and receive queues. The receive queues are the odd numbers. Divide them by 2 (discard the remainder) to get the queue number. i.e. vring 1 is queue 0. ready state 1 = enabled. ready state 0 = disabled. In this example, the vSRX is only enabling 4 queues:

```
 2020-01-22 14:02:44,060 UVHOST: Client _tap4966ea8d-49: setting vring 0 ready state 1
 2020-01-22 14:02:44,060 UVHOST: Client _tap4966ea8d-49: setting vring 1 ready state 1
 2020-01-22 14:02:44,060 UVHOST: Client _tap4966ea8d-49: setting vring 2 ready state 1
 2020-01-22 14:02:44,060 UVHOST: Client _tap4966ea8d-49: setting vring 3 ready state 1
 2020-01-22 14:02:44,060 UVHOST: Client _tap4966ea8d-49: setting vring 4 ready state 1
 2020-01-22 14:02:44,060 UVHOST: Client _tap4966ea8d-49: setting vring 5 ready state 1
 2020-01-22 14:02:44,060 UVHOST: Client _tap4966ea8d-49: setting vring 6 ready state 1
 2020-01-22 14:02:44,060 UVHOST: Client _tap4966ea8d-49: setting vring 7 ready state 1
 2020-01-22 14:02:44,061 UVHOST: Client _tap4966ea8d-49: setting vring 8 ready state 0
 2020-01-22 14:02:44,061 UVHOST: Client _tap4966ea8d-49: setting vring 9 ready state 0
```

Packet Flow Summary (Based on R1910)

Packet Origination	Packet Type	Action
pNIC	IPv4 MPLSoUDP or VXLAN	Processed by polling core
	IPv4 MPLSoGRE - Inner packet is IPv4 or IPv6	Distributed to a different processing core based on hash of inner packet.
	IPv4 MPLSoGRE - Inner packet not IPv4 or IPV6	Processed by polling core.
vNIC	IPv4 MPLSoGRE - Inner packet is iPv4 or IPV6	Distributed to a different processing core based on hash of inner packet.
	IPv4 MPLSoGRE - Inner packet not IPv4 or IPv6	Processed by polling core
	All other IPv4 packets	Distributed to a different processing core based on hash of packet.
	All IPv6 packets	Distributed to a different processing core based on hash of packet.
	All non-IPv4/IPv6 packets (i.e. L2 packets)	Processed by polling core

Note that whenever the polling core performs a hash to distribute the packet to a processing core, the polling core will never select itself to process the packet. The selection of available processing cores for each polling core can be seen in the /var/log/containers/contrail/contrail-vrouter-dpdk.log file. It mentions MPLSoGRE, but this applies to all packets that are distributed via hashing by the polling core:

```
2020-01-07 13:08:01,403 VROUTER: Lcore 10: distributing MPLSoGRE packets to [11,12,13,14,15]
2020-01-07 13:08:01,403 VROUTER: Lcore 12: distributing MPLSoGRE packets to [10,11,13,14,15]
2020-01-07 13:08:01,403 VROUTER: Lcore 14: distributing MPLSoGRE packets to [10,11,12,13,15]
2020-01-07 13:08:01,403 VROUTER: Lcore 11: distributing MPLSoGRE packets to [10,12,13,14,15]
2020-01-07 13:08:01,404 VROUTER: Lcore 13: distributing MPLSoGRE packets to [10,11,12,14,15]
2020-01-07 13:08:01,404 VROUTER: Lcore 15: distributing MPLSoGRE packets to [10,11,12,13,14]
```

Non DPDK VM

When sending packets from NIC to non-DPDK VM, DPDK vRouter raises an interrupt in the guest. This is an additional step after copying the packet to that VM. The interrupt is only needed because the VM is not polling for packets. The vRouter writes to a file descriptor, which tells the kernel to raise an interrupt to non-DPDK VM. The file descriptor is sent by Qemu to vrouter when the VM is spawned. Also note that an Interrupt is raised for a burst of packets, not for every packet.

To avoid interrupts raised by DPDK vRouter (required to raise packet processing by QEMU/KVM) that influence packets processing Juniper recommends to only connect DPDK VMs and DPDK interfaces to DPDK vRouter.

Putting non-DPDK VMs on a DPDK node is affecting performance of both VM and vRouter:

- VM performance is impacted because each interrupt raised would cause a “VMExit” (it has a very bad impact on performance)
- vRouter performance is also impacted since it needs to do additional work of “raising an interrupt” after it enqueues the packet to the vNIC Ring.

As a result of these, the performance numbers would be the similar as that of kernel vRouter.

Step by Step vif (vhost-user) setup and non DPDK VM packet processing:

1. When the VM is spawned, QEMU registers an fd (file descriptor) for the guest (irqfd) and passes to the backend (vRouter). The guest listens to this fd for changes to process the packets.

2. registration mechanism for this fd is provided by KVM kernel module. This is KVM module that actually provides a wait-notify mechanism between the guest and the backend (vRouter)

3. virtual machine is setting a flag “VRING_AVAIL_F_NO_INTERRUPT” into VirtIO ring by which vRouter will be aware it needs to notify the Guest VM that a packet has been delivered (this flag is not set when VM is using DPDK).

4. once the vRouter gets hold of this fd, all it needs to do is to enqueue the packets to the virtio ring and write to that fd.

5. KVM injects this event to the VM as an interrupt. In this process, the VM needs a 'VMExit'

6. VMExit latency is very high (around 10K clock cycles). This is further compounded by the NAPI latency. It has a very bad impact on VM performance. vRouter performance is also impacted due to the extra work of writing to the fd (which interrupts the guest).

Supported NICs

NICs	Ubuntu - KVM	Ubuntu - DPDK	Redhat - KVM	Redhat - DPDK	vCenter - ESX
Intel 82599/X520 “Niantic” - 10G	Yes	Yes	Yes	Yes	Yes
Intel X710 “Fortville” - 10G/25G/40G	Yes	Yes	Yes	Yes	Not tested
Broadcom bnxt 2x25G	Yes	Yes	Not tested	Not tested	Not tested
Mellanox 2x25G	Yes	Yes	Not tested	Not tested	Not tested
Netronome	3.1.x only	No	No	No	No

These are the guidelines from Intel with regard to Intel NIC X710 (in order not to bump into a known issue that impacts data plane)

- Do not use a single PCI NIC for non-DPDK and DPDK interfaces (having i40e kernel driver used together with i40e PMD driver for interfaces on the same PCI NIC cause problems)
- Upgrade NIC FW to 6.01 and i40e kernel driver to 2.4.6 (all necessary information you find in that document, table 9. https://www.intel.com/content/dam/www/public/us/en/documents/release-notes/xl710-ethernet-controller-feature-matrix.pdf)
  NOTE: If your server manufacturer has no support for the latest firmware please contact him asap (i.e. The latest Lenovo server firmware includes 5.05 for X710 NICs)
- LLDP is supported from 6.01 NIC FW but Intel also suggested to disable (ethtool –set-priv-flags <interface name> disable-fw-lldp on)

Redhat supports 1.6.27 i40 kernel driver version. Canonical supports 1.4.25 i40e kernel version.

Contrail related linux packages

Ubuntu

Package name	Description
contrail-dpdk-kernel-modules-dkms	Contrail DPDK 17.02 library in DKMS format
contrail-vrouter-dpdk	Core vRouter DPDK binary
contrail-vrouter-dpdk-init	DPDK compute-node startup and monitoring scripts like: contrail-vrouter-dpdk.ini dpdk_nic_bind.py if-vhost0

Redhat

Package name	Description
dpdk	RHEL DPDK package
contrail-vrouter-dpdk	Core vRouter DPDK binary
contrail-vrouter-dpdk-init	DPDK compute-node startup and monitoring scripts like: contrail-vrouter-dpdk.ini dpdk_nic_bind.py if-vhost0

Note: info how to deploy with Director with Openstack Newton
https://github.com/Juniper/contrail-tripleo-heat-templates/tree/stable/newton

Note2: DPDK version used by the vrouter can be checked into dpdk vrouter log files :

```
$ vi /var/log/containers/contrail/contrail-vrouter-dpdk.log
...
2019-04-19 16:30:41,411 VROUTER: vRouter version: {"build-info": [{"build-time": "2019-04-11 23:47:32.407280", "build-hostname": "rhel-7-builder-juniper-contrail-ci-c-0000225573.novalocal", "build-user": "zuul", "build-version": "5.1.0"}]}
2019-04-19 16:30:41,411 VROUTER: DPDK version: DPDK 18.05.1
...
```

Nova / Qemu / Libvirt

It is required to have Contrail qemu / libvirt / nova packages installed on DPDK nodes in order to:

- Support multiqueue (Multiqueue support is available from Mitaka release)
- Not lose connectivity to VMs when restarting DPDK vrouter

PS: This limitation is removed.It is possible to use qemu and nova upstream packages with contrail-nova-vif-driver. Starting from OpenStack Pike release, no patch is required. libvirt is the only package needed if we want multiqueue.

Configuration description

Compute node memory and huge pages

Huge pages configuration

DPDK vrouter is using hugepages. Huge pages need to be configured before it can be used. Only smaller (2MB) pages can be configured using sysctl. 1GB pages are strongly recommended, so the only consistent way of configuration is to add parameters to linux kernel.

NOTE: When using 2MB hugepages, the amount configured cannot be bigger than 32768. If more pages are configured, vrouter-dpdk will not start. This is a limitation of DPDK library.

Allocating huge pages should be done based on the prediction of how many VMs (their amount of memory) will be used on a hypervisor + 2GB for vRouter for default configuration. Of course some memory need to be left for operating system and other processes.

NOTE: In RHEL environment, to make vrouter-dpdk work with 1G hugepages, a small amount of 2MB hugepages needs to be declared too (128 pages is enough).

The following kernel parameters configure huge pages:

```
default_hugepagesz=1GB hugepagesz=1G hugepages=40 hugepagesz=2M
hugepages=40
```

`default_hugepagesz` defines which huge page size is a default (this size will appear in /proc/meminfo, and this size will be mounted by default when pagesize mounting option will not be used)

`hugepagesz` followed by `hugepages` defines size and amount respectively and the pair can be repeated to configure different sizes of huge pages.

After adding the configuration to a kernel and rebooting server, the number of allocated huge pages can be checked from command line. The most common way of doing it is to check /proc/meminfo. This is not a good way, because it shows only one (default) huge page size.

Since it is possible to configure multiple sizes of hugepages, the only source of information is /sys filesystem (for each numa node and each size - total/free):

```
# cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
# cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/free_hugepages
# cat /sys/devices/system/node/node0/hugepages/hugepages-1048576kB/nr_hugepages
# cat /sys/devices/system/node/node0/hugepages/hugepages-1048576kB/free_hugepages
# cat /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr_hugepages
# cat /sys/devices/system/node/node1/hugepages/hugepages-2048kB/free_hugepages
# cat /sys/devices/system/node/node1/hugepages/hugepages-1048576kB/nr_hugepages
# cat /sys/devices/system/node/node1/hugepages/hugepages-1048576kB/free_hugepages
```

To finish configuration of hugepages, hugetlbfs pseudo filesystem needs to be mounted. The following line needs to be added to /etc/fstab:

```
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,seclabel)
```

As mentioned before, DPDK vRouter needs its own hugepages. The DPDK library auto-detects the huge pages from the hugetlbfs mount point. 2GB for vRouter are based on setup provided in configuration file.

```
/etc/contrail/supervisord_vrouter_files/contrail-vrouter-dpdk.ini
[program:contrail-vrouter-dpdk]
command=/usr/bin/taskset 0x154000000154 /usr/bin/contrail-vrouter-dpdk --no-daemon
--vr_flow_entries=2000000 --vdev
"eth_bond_bond0,mode=4,xmit_policy=l34,socket_id=0,mac=90:e2:ba:c5:79:90,slave=0000:01:00.0,slave=
0000:01:00.1,slave=0000:02:00.0,slave=0000:02:00.1" --socket-mem 1024,1024
```

--socket-mem 1024,1024 - means allocate 1GB memory (in huge pages) per NUMA node for vRouter (here we assume 2 NUMA nodes). Even if vrouter is only pinned on a single node, memory has to be allocated on both as virtual machines are pinned on all nodes and thus will require memory on each.

Hugepage memory and NUMA sockets

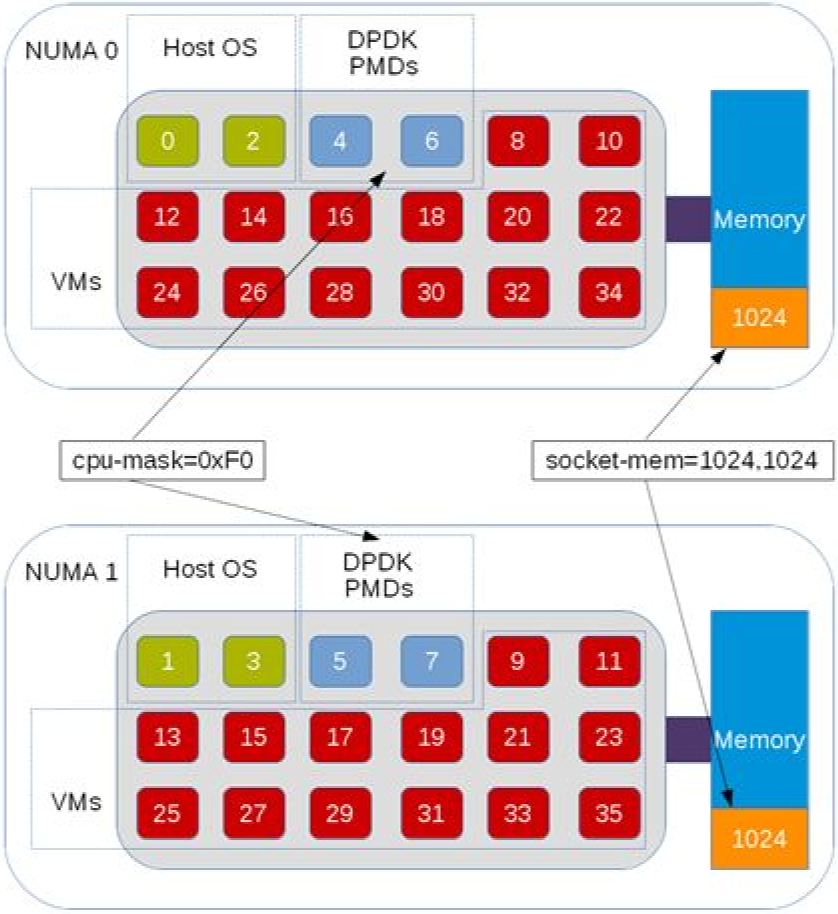
It is important to allocate hugepage memory to all NUMA nodes that will have DPDK interfaces associated with them. If memory is not allocated on a NUMA node associated with a physical NIC or VM, they cannot be used. If you are using 2 or more ports from different NICs, it is best to ensure that these NICs are on the same CPU socket.

In order to allocate memory on the first NUMA socket, we are using option with only one parameter:

```
--socket-mem <value>
```

In order to allocate memory on the NUMA0 and NUMA1 socket, we are using option with only two parameters:

```
--socket-mem <value>,<value>
```



PS: On a 2 NUMA nodes we have to allocate memory on each of them even if vRouter is only pinned on a single one. Indeed Virtual machines are pinned on both and require memory on each.

We can get all PCI Ethernet devices installed on the host using the following command :

```
# lspci -nn | grep Eth
18:00.0 Ethernet controller [0200]: Intel Corporation 82599 10 Gigabit Dual Port Backplane Connection [8086:10f8] (rev 01)
18:00.1 Ethernet controller [0200]: Intel Corporation 82599 10 Gigabit Dual Port Backplane Connection [8086:10f8] (rev 01)
5e:00.0 Ethernet controller [0200]: Intel Corporation 82599 10 Gigabit Dual Port Backplane Connection [8086:10f8] (rev 01)
5e:00.1 Ethernet controller [0200]: Intel Corporation 82599 10 Gigabit Dual Port Backplane Connection [8086:10f8] (rev 01)
```

We can check the PCI device related numa node id using one of following commands:

```
$ lspci -vmms 18:00.0 | grep NUMANode
NUMANode:     0

$ cat /sys/bus/pci/devices/0000\:18\:00.0/numa_node
0
```

Using numactl we can get CPU IDs on each NUMA socket:

```
# numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70
node 0 size: 96965 MB
node 0 free: 10842 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71
node 1 size: 98304 MB
node 1 free: 12845 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

PS: When DPDK vrouter is used, OpenStack flavors must have `hw:mem_page_size` property (cf OpenStack configuration section).

Contrail vrouter and Single Hugepage size (Kubernetes)

Currently Kubernetes in not supporting to be run on compute nodes configured with multiple huge page size. If a worker node is configured with both 2M and 1GB hugepage size, Kubelet will fail to start.

Contrail DPDK vrouter is able to run with a single Hugepage size and does not required to get both 2M and 1G pages allocated. But per default, Contrail vrouter assumes that provided hugepages are 2Mb.

If only 1G huge pages are configured on the compute node, pagesize parameter must be specified into huge page table mount point in order for the vrouter to be aware that only 1G huge pages are available on the compute node :.
Cf: https://github.com/Juniper/contrail-vrouter/blob/R1908/dpdk/vr_dpdk_table_mem.c#L80-L84

If this parameter is not present in the mountpoint, Contrail vrouter is assuming that 2MB hugepages are available, and will requests 2M huge pages. If only 1GB huge pages are available onto the compute node, vrouter will fail to start.

Hugepage size not specified in mount point
When huge pages mount point is configured without pagesize parameter, vrouter will request 2M hugepages allocation.

/etc/fstab:

```
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,seclabel)
```

```
# cat /proc/mounts | grep hugepage
hugetlbfs /dev/hugepages hugetlbfs rw,seclabel,relatime 0 0
```

1G hugepage size not specified in mount point
Here we have specified huge pages mount with pagesize=1G parameter. vrouter will request 1G hugepages allocation at startup.

/etc/fstab:

```
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,seclabel,pagesize=1G)
```

```
# cat /proc/mounts | grep hugepage
hugetlbfs /dev/hugepages hugetlbfs rw,seclabel,relatime,pagesize=1G 0 0
```

Compute node processing and CPU allocation

Physical CPU cores allocation planning

When planning physical CPU cores allocation, the following aspects need to be taken into account:

- hyperthreading enabled or not
- numa topology
- number of cores assigned to vrouter for 2 kinds of tasks:
    - forwarding threads (packet processing purpose)
    - control and service threads (vrouter management purpose)
- number of cores left for system processes
- number of cores allocated to VMs

WARNING: A proper definition and configuration of CPU partitioning is key for optimal performance. A bogus implementation is indeed the main source of transient packet drops even at moderate throughput.

Hyper Threading

If HT (Hyper Threading) is enabled, the first half of each numa node core are physical cores, and the second half their HT sibling. The best way to utilize all cores in the system, especially when using *hw:cpu_policy=dedicated* (see Linux kernel isolcpus option) is to **use both siblings when calculating mask for vrouter (and other resources line systemd)**. In that case, both siblings for each core used for VM pinning can be utilized.

With OpenStack Having a core without its sibling listed in *vcpu_pin_set* variable in *nova.conf* file and *hw:cpu_policy=dedicated* set in flavor properties used to spawn an instance leads to a scheduling error (variable ‘sibling_set’ referenced before assignment).

CPU assignment and isolation mechanisms

Tasks to be run by an operating system must be spread across available CPUs. These tasks into a multi-threading environment are often made of several processes which are also made of several threads. In order to run all these processes and threads on the CPU offered by the node, the Operating System is using a scheduler to place each single one onto a given CPU. There are two styles of scheduling, cooperative and preemptive. By default, RedHat Linux is using a cooperative mode.
Cf: https://medium.com/traveloka-engineering/cooperative-vs-preemptive-a-quest-to-maximize-concurrency-power-3b10c5a920fe

In order to get a CPU booked for a subset of tasks, we have to inform the Operating System scheduler not to use these CPUs for all the tasks it has to run. These CPUs are told: *"isolated"* because they are no more used to process all tasks. In order to get a CPU isolated several mechanisms can be used:

- remove this CPU from the "common" CPU list used to process all tasks
- change the scheduling algorithm (cooperative to preemptive)
- participate or not to interrupt processing

RedHat Linux is currently supporting 2 different CPU isolation methods. They are *isolcpus* and *tuned CPU partitioning*, They can be used either independently or simultaneously.

A first method, ***isolcpus*** kernel parameter, has been proposed to keep CPUs away from linux scheduler. This isolation mechanism will:

- remove isolated CPUs from the "common" CPU list used to process all tasks
- change the scheduling algorithm from cooperative to preemptive
- perform CPU isolation at the system boot

The main drawbacks of using *isolcpus* are :

- it requires manual placement of processes on isolated cpus.
- it is not possible to rearrange the CPU isolation rules after the system startup
- it is not possible to move process from one isolated cpu to another.

A second method, ***tuned CPU partitioning***, has been proposed more recently. since almost all processes are started by systemd, physical CPU cores assignment can be configured in a different way:

- *CPUAffinity* parameter in /etc/systemd/system.conf; it will restrict all processes spawned by systemd to the list of cores. Note that from RHEL 7.5, CPUAffinity is natively integrated in *tuned's cpu-partitioning profile* (more information on this topic is provided later in this section).
- *isolated_cores*: is removing a set of CPU from the "common" CPU list used to process all tasks

This isolation mechanism will:

- remove isolated CPUs from the "common" CPU list used to process all tasks
- perform CPU isolation after the system boot using systemd.

The main drawbacks of using *tuned partitioning* are :

- Some processes are started at the system boot before systemd is started. These processes are run before isolation rules are enforced and could break expected isolation rules.
- Scheduling Algorithm is kept to cooperative mode which provides a lower isolation.

In few words, tuned partitioning is more flexible and featureful than isolcpus, but is providing a lower CPU isolation. This is why both mechanisms are often used altogether to enforce CPU isolation.

IMPORTANT NOTE: if both tuned CPU partitioning profiles and isolcpus mechanisms are used **be careful to be consistent**. in the defined setup.

isolcpus mechanism configuration

WARNING: before Contrail 20.03 release, it is not recommended to use such an isolation method. If used some packet drops could randomly occur and vrouter performances are not stable.

isolcpus is a kernel parameter.It has to be provisioned at the system startup. GRUB configuration has to be modified in order to take into consideration a new set of isolated CPU, and then, the system has to be restarted.

```
$ vi /etc/default/grub
GRUB_CMDLINE_LINUX="console=tty0 console=ttyS0,115200n8 crashkernel=auto rhgb quiet
default_hugepagesz=1GB hugepagesz=1G hugepages=28 iommu=pt intel_iommu=on
isolcpus=7,9-35,43,45-71"
$ grub2-mkconfig -o /etc/grub2.cfg
```

When TripleO is used for Contrail and OpenStack installation, this grub configuration update is automatically done, using TRIPLEO_HEAT_TEMPLATE_KERNEL_ARGS environment variable to define i*solcpus* value.

For instance :

```
TRIPLEO_HEAT_TEMPLATE_KERNEL_ARGS: “isolcpus=7,9-35,43,45-71”
```

Tuned CPU partitioning configuration

Tuned cpu partitioning has to be installed onto the system:

```
yum install tuned-profiles-cpu-partitioning
```

Tuned is using CPU isolation information defined into cpu-partitioning-variables.conf and into system.conf:

/etc/tuned/cpu-partitioning-variables.conf
isolated_cores=7,9-35,43,45-71

/systemd/system.conf
CPUAffinity=0-6,8,36-42,44

Tuned is not setting *isolcpus* Kernel value, but *tuned.non_isolcpus*.

DPDK vrouter process and threads

DPDK vrouter forwarding plane process is made up of several kinds of threads:

- packet processing threads: used to perform packet switching
- control and service threads: used for DPDK vrouter configuration (add/remove vif interfaces onto the vrouter, manage the communication with the vrouter agent for instance)

Each set of threads is made up of several single threads:

- **control threads** : eal-intr-thread, rte_mp_handle, rte_mp_async (they are generated by the DPDK library itself - DPDK setup management)
- **service threads** : thread names are lcore 0 to 9. They each one has a specific role:
  - lcore 0: vhost0
  - lcore 1: timers
  - lcore 2: Interrupts
  - lcore 8: pkt0 (can be busy)
  - lcore 9: Netlink (can be busy)
- **processing threads** : thread names are lcore 10 and above. All the threads are used for packet processing (polling and forwarding)

Service and Processing threads are *named lcore-slave-**<ID>***. Contrail vRouter cores ID have a specific meaning defined in the following C enum data structure:

```
enum {
      VR_DPDK_KNITAP_LCORE_ID = 0,
      VR_DPDK_TIMER_LCORE_ID,
      VR_DPDK_UVHOST_LCORE_ID,
      VR_DPDK_IO_LCORE_ID,     = 3
      VR_DPDK_IO_LCORE_ID2,
      VR_DPDK_IO_LCORE_ID3,
      VR_DPDK_IO_LCORE_ID4,
      VR_DPDK_LAST_IO_LCORE_ID,     # 7
    VR_DPDK_PACKET_LCORE_ID,      # 8
    VR_DPDK_NETLINK_LCORE_ID,
      VR_DPDK_FWD_LCORE_ID,         # 10
};
```

We can find those names by running using the “ps” command with some additional arguments (this is from a compute node where vrouter is using 8 logical forwarding cores, 4 phy cores using HT siblings):

```
# ps -T -p `pidof contrail-vrouter-dpdk`
  PID  SPID TTY          TIME CMD
54490 54490 ?      02:46:12 contrail-vroute
54490 54611 ?      00:02:33 eal-intr-thread
54490 54612 ?      01:35:26 lcore-slave-1
54490 54613 ?      00:00:00 lcore-slave-2
54490 54614 ?      00:00:17 lcore-slave-8
54490 54615 ?      00:02:14 lcore-slave-9
54490 54616 ?      2-21:44:06 lcore-slave-10
54490 54617 ?      2-21:44:06 lcore-slave-11
54490 54618 ?      2-21:44:06 lcore-slave-12
54490 54619 ?      2-21:44:06 lcore-slave-13
54490 54620 ?      2-21:44:06 lcore-slave-14
54490 54621 ?      2-21:44:06 lcore-slave-15
54490 54622 ?      2-21:44:06 lcore-slave-16
54490 54623 ?      2-21:44:06 lcore-slave-17
54490 54990 ?      00:00:00 lcore-slave-9
```

Here we have :

- contrail-vrouter is main thread
- lcore-slave-1 is timer thread
- lcore-slave-2 is uvhost (for qemu) thread
- lcore-slave-8 is pkt0 thread
- lcore-slave-9 is netlink thread (for nh/rt programming)
- lcore-slave-10 onwards are forwarding threads, the ones running at 100% as they are constantly polling the interfaces

To list all lightweight processes created by contrail-vrouter-dpdk run following command:

```
# pstree -p $(ps -ef | awk '$8=="/usr/bin/contrail-vrouter-dpdk" {print $2}')
contrail-vroute(6665)─┬─{contrail-vroute}(7800)
                      ├─{contrail-vroute}(7801)
                      ├─{contrail-vroute}(7802)
                      ├─{contrail-vroute}(7803)
                      ├─{contrail-vroute}(7804)
                      ├─{contrail-vroute}(7805)
                      ├─{contrail-vroute}(7806)
                      ├─{contrail-vroute}(7807)
                      ├─{contrail-vroute}(7808)
                      └─{contrail-vroute}(8200)
```

The following command can be used. It provides in the last column the CPU load generated by each thread:

```
ps -mo pid,tid,comm,psr,pcpu -p $(ps -ef | awk
'$8=="/usr/bin/contrail-vrouter-dpdk" {print $2}')

PID TID COMMAND PSR %CPU
161791 - contrail-vroute - 618
- 161791 - 22 1.6
- 161867 - 3 0.0
- 161868 - 27 1.1
- 161869 - 18 0.0
- 161870 - 27 0.3
- 161871 - 21 0.0
- 161872 - 2 64.3
- 161873 - 3 64.3
- 161874 - 4 64.3
- 161875 - 5 64.3
- 161876 - 6 64.3
- 161877 - 7 98.3
- 161878 - 8 98.1
- 161879 - 9 97.7
- 162134 - 11 0.0
```

Using ***pidstat*** command we can see that some vrouter threads are running at 100% CPU. Those are the forwarding threads, the ones pinned based on the configured coremask :

```
# pidstat -t -p `pidof contrail-vrouter-dpdk`
Linux 3.10.0-957.10.1.el7.x86_64 (compute_test) 14/10/2019      _x86_64_        (48 CPU)
10:02:46      UID      TGID       TID    %usr %system  %guest    %CPU CPU  Command
10:02:46        0     21666         -  100,00  100,00    0,00  100,00 25 contrail-vroute
10:02:46        0         -     21666    0,47    0,53    0,00    1,00 25 |__contrail-vroute
10:02:46        0         -     21740    0,00    0,00    0,00    0,00 28 |__rte_mp_handle
10:02:46        0         -     21741    0,00    0,00    0,00    0,00 28 |__rte_mp_async
10:02:46        0         -     21790    0,00    0,00    0,00    0,00 24 |__eal-intr-thread
10:02:46        0         -     21791    1,37    0,41    0,00    1,78 1 |__lcore-slave-1
10:02:46        0         -     21792    0,00    0,00    0,00    0,00 0 |__lcore-slave-2
10:02:46        0         -     21793    0,00    0,01    0,00    0,01 0 |__lcore-slave-8
10:02:46        0         -     21794    0,06    0,00    0,00    0,06 0 |__lcore-slave-9
10:02:46        0         -     21795   52,83   47,01    0,00   99,84 2 |__lcore-slave-10
10:02:46        0         -     21796   54,07   45,82    0,00   99,88 4 |__lcore-slave-11
10:02:46        0         -     21797   52,66   47,24    0,00   99,90 6 |__lcore-slave-12
10:02:46        0         -     21798   52,71   47,20    0,00   99,91 8 |__lcore-slave-13
10:02:46        0         -     21799   52,56   47,37    0,00   99,93 26 |__lcore-slave-14
10:02:46        0         -     21800   52,35   47,58    0,00   99,93 28 |__lcore-slave-15
10:02:46        0         -     21801   52,40   47,54    0,00   99,94 30 |__lcore-slave-16
10:02:46        0         -     21802   52,69   47,25    0,00   99,94 32 |__lcore-slave-17
10:02:46        0         -     29401    0,00    0,00    0,00    0,00 0 |__lcore-slave-9
```

In the example above, we can notice:

- **processing threads** in blue, we almost 100% CPU load.
- **service threads** in green.
- **control threads** in red.

In order to list CPU cores assigned to contrail-vrouter-dpdk, we have to run taskset command for each lightweight process.

```
# taskset -cp 21791
pid 21666's current affinity list: 0-47

# taskset -cp 21795
pid 21795's current affinity list: 2
```

PS: pidstat command is not provided in default OS installation. It has to be installed with following command on RedHat system:
# yum install -y sysstat

CPU assignment for DPDK vRouter

In order to get DPDK vrouter threads pinned to a subset of CPUs on each compute node we have to define:

- vrouter dpdk cores used for ***packet processing*** (polling and forwarding threads) in ***CPU_LIST*** (or coremask - cf further explanations) within following file in:
    - /etc/contrail/supervisord_vrouter_files/contrail-vrouter-dpdk.ini (up to 4.1)
    - /etc/sysconfig/network-scripts/ifcfg-vhost0 (from 5.0 to higher)
- vrouter dpdk cores used for ***vrouter management*** (services threads) in ***SERVICE_CORE_MASK*** within following file: /etc/sysconfig/network-scripts/ifcfg-vhost0 (from 20.03 to higher)
- vrouter dpdk cores used for ***vrouter management*** (DPDK control threads) in ***DPDK_CTRL_THREAD_MASK*** within following file: /etc/sysconfig/network-scripts/ifcfg-vhost0 (from 20.03 to higher)

These values: CPU_LIST, DPDK_CTRL_THREAD_MASK and SERVICE_CORE_MASK are CPU masks used for CPU pinning with taskset command.

CPU assignment for DPDK vRouter packet processing threads

vRouter ***CPU_LIST*** (also named ***coremask*** in some configuration files) variable is allowing to define which core CPU will be allocated for router forwarding and polling processing threads.

vrouter forwarding and polling processings are CPU intensive. vrouter logical threads named logica core ID 10 to logical core 10 + N - 1 (with N = Number of allocated polling and forwarding cores) are used for forwarding and polling processing. These threads have to be pinned to some well defined CPU in order to avoid the vrouter to starve the full node CPU capacity. A maximum of 54 forwarding CPU can be allocated to a vrouter (logical core ID: 10 to 63).

Below is an example of a configuration of a 2 socket system, each processor with 2*18 physical cores, with HT enabled. The network adapter in PCI-E bus is attached to NUMA 0. 4 physical cores are dedicated for vRouter (i.e. 8 logical cores thanks to Hyper Threading).

```
NUMA node0 CPU(s):
      PHY cores: 0  2  4  6  8  10 12 14 16 18 20 22 24 26 28 30 32 34
      HT cores : 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70
NUMA node1 CPU(s):
      PHY cores: 1  3  5  7  9  11 13 15 17 19 21 23 25 27 29 31 33 35
      HT cores : 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71
```

We are planning CPU assignment as follows:

- 4 cores for Operating System processes and vrouter agent/dpdk lightweight threads: **red 0, 1, 3, 5, 36, 37, 39, 41**
- 4 physical cores for vRouter DPDK forwarding threads (in same NUMA 0), **blue**: **2, 4, 6, 8** with HT siblings **38, 40, 42, 44**. Do not allocate physical Core 0 (e.g. here 0 and 36) for Vrouter DPDK forwarding threads.
- The rest of the cores are for Nova vcpu_pin_set, **black**: 7,9-35, 43,45-71. For optimal performance it is recommended that a single VMs is executed on a same NUMA (unless it can efficiently leverage kvm/pinning information)

vRouter **CPU_LIST** (aka coremask) calculation based on the assumption above: vCPUs 2, 4, 6, 8, 38, 40, 42, 44 are allocated for vRouter, which maps to b0001 0101 0100 0000 0000 0000 0000 0000 0000 0001 0101 0100 in binary.

This translates to a hex mask 0x154000000154 that must be configured in the /etc/contrail/supervisord_vrouter_files/contrail-vrouter-dpdk.ini file (see below example for Contrail 4.1 release).

```
[root@overcloud-contraildpdk-16 ~]# cat
/etc/contrail/supervisord_vrouter_files/contrail-vrouter-dpdk.ini
[program:contrail-vrouter-dpdk]
command=/bin/taskset 0x154000000154 /usr/bin/contrail-vrouter-dpdk
--no-daemon --vdev
"eth_bond_bond1,mode=4,xmit_policy=l34,socket_id=0,mac=a0:36:9f:d1:c8:78,sl
ave=0000:04:00.0,slave=0000:04:00.1" --vlan_tci "201" --vlan_fwd_intf_name
"bond1" --socket-mem 1024,1024
priority=410
autostart=true
killasgroup=true
stdout_capture_maxbytes=1MB
redirect_stderr=true
stdout_logfile=/var/log/contrail/contrail-vrouter-dpdk-stdout.log
stderr_logfile=/var/log/contrail/contrail-vrouter-dpdk-stderr.log
exitcodes=0                   ; 'expected' exit codes for process (default
0,2)
```

After modifying this file, the vrouter must be restarted to take changes into account:

```
systemctl restart supervisor-vrouter
```

NOTE: Some processors have different layout of cores in each numa node. The most popular is the first half being a part of numa 0 and the second half a part of numa 1. But there are processors, where even cores are a part of numa 0 and odd cores a part of numa

NOTE2: When using 5.0 and later version of contrail vrouter, hex mask 0x154000000154, must be configured into CPU_LIST variable in /etc/sysconfig/network-scripts/ifcfg-vhost0 file:

```
CPU_LIST=0x154000000154
```

After modifying this file, the vrouter vhost0 interface must be restarted to take changes into account:

```
$ ifdown vhost0
$ ifup vhost0
```

CPU assignment for DPDK vRouter service and control threads

contrail version <= 19.12

For contrail release 19.12 and earlier, Control and Service threads are not assigned to any CPU. Consequently, these threads can use any available CPU.

We can get all contrail-vrouter-dpdk assignments using taskset with -a option :

```
# taskset -cap `pidof contrail-vrouter-dpdk`
pid 21666's current affinity list : 0-71
pid 21740's current affinity list : 3-7,18-71
pid 21741's current affinity list : 3-7,18-71
pid 21790's current affinity list : 3-7,18-71
pid 21791's current affinity list : 0-71
pid 21792's current affinity list : 0-71
pid 21793's current affinity list : 0-71
pid 21794's current affinity list : 0-71
pid 21795's current affinity list : 2
pid 21796's current affinity list : 4
pid 21797's current affinity list : 6
pid 21798's current affinity list : 8
pid 21799's current affinity list : 38
pid 21800's current affinity list : 40
pid 21801's current affinity list : 42
pid 21802's current affinity list : 44
pid 29401's current affinity list : 0-71
```

In the previous example, we can notice that threads are assigned to a given CPU set according following rules :

- **processing threads** have a CPU affinity restricted to a single one CPU (these are 2,4,6,8,38,40,42 and 44). Consequently each is placed on a single CPU.
- **service threads** have a CPU affinity with all available CPUs. They can use any CPU..
- **control threads** have CPU affinity with almost all available CPUs. This is the default DPDK library behavior (DPDK is keeping only CPU that are not booked for vrouter DPDK processing threads - previous example these are 3-7,18-71).

PS: It’s possible for a Contrail release R1912 (and earlier) to apply a patch in order to be able to use a specific *SERVICE_CORE_LIST* parameter. This parameter allows to pin both service and control threads onto a specific CPU list. In order to get the new parameter available, we have to apply the following procedure on all the DPDK compute node :

```
# mkdir -p /tmp/vrouter_patch
# cd /tmp/vrouter_patch
# curl -o patched-r1912-archive.tar.gz \
 'https://review.opencontrail.org/changes/Juniper%2Fcontrail-container-builder~55730/revisions/1/archive?format=tgz'
# tar xvzf ./patched-r1912-archive.tar.gz containers/vrouter/base/network-functions-vrouter-dpdk
containers/vrouter/agent-dpdk/entrypoint.sh  --strip 3
# mv ./network-functions-vrouter-dpdk /etc/sysconfig/network-scripts/network-functions-vrouter-dpdk
# X=$(docker images --format "{{.Repository}}:{{.Tag}}" | grep contrail-vrouter-agent-dpdk)
# cat <<EOF > Dockerfile
# FROM ${X}
# COPY ./entrypoint.sh /entrypoint.sh
# EOF
# docker build . -t ${X}
```

SERVICE_CORE_LIST has been split into two variables into Contrail 20.03 and later releases:

- DPDK_CTRL_THREAD_MASK
- SERVICE_CORE_MASK

In order to be used, you’ll have to follow explanation in next section given for SERVICE_CORE_MASK (just replace in you configuration SERVICE_CORE_MASK with SERVICE_CORE_LIST when using this patch)

vRouter control and service threads pinning - contrail version >= 20.03

Since Contrail 20.03 release it is possible to assign Control and Service threads to a given CPU list. vRouter ***SERVICE_CORE_MASK*** variable is allowing to define which core CPU will be allocated for vrouter service threads. ***DPDK_CTRL_THREAD_MASK*** variable is allowing to define which core CPU will be allocated for vrouter control threads.

Below is an example of a configuration of a 2 socket system, each processor with 2*18 physical cores, with HT enabled. The network adapter in PCI-E bus is attached to NUMA 0. 4 physical cores are dedicated for vRouter (i.e. 8 logical cores thanks to Hyper Threading).

```
NUMA node0 CPU(s):
     PHY cores: 0  2  4  6  8  10 12 14 16 18 20 22 24 26 28 30 32 34
     HT cores : 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70
NUMA node1 CPU(s):
     PHY cores: 1  3  5  7  9  11 13 15 17 19 21 23 25 27 29 31 33 35
     HT cores : 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71
```

We are planning CPU assignment as follows:

- 4 cores for Operating System processes and vrouter agent/dpdk lightweight threads: **red 0, 1, 3, 5, 36, 37, 39, 41.** DPDK control threads will also be pinned on these CPUs.
- 4 physical cores for vRouter DPDK vrouter forwarding threads (in same NUMA 0), **blue**: **2, 4, 6, 8** with HT siblings **38, 40, 42, 44**. Do not allocate physical Core 0 (e.g. here 0 and 36) for Vrouter DPDK threads.
- 1 physical core for vRouter DPDK vrouter service threads (in same NUMA 0), **green**: **10** with HT sibling **46**.
- The rest of the cores are for Nova vcpu_pin_set, **black**: 7,9,11-35, 43,45,47-71. For optimal performance it is recommended that each VM be executed on a same NUMA (unless it can efficiently leverage kvm/pinning information).

vRouter service cores mask calculation based on the assumption above: vCPUs 10, 46 are allocated for vRouter, which maps to b0100 0000 0000   0000 0000 0000 0000 0000 0000 0100 0000 0000 in binary.

This translates to a hex mask 0x400000000400 that must be configured into **SERVICE_CORE_MASK** variable in /etc/sysconfig/network-scripts/ifcfg-vhost0 file:
SERVICE_CORE_MASK=**`0x400000000400`**

vRouter DPDK control threads mask calculation based on the assumption above: vCPUs 0,1,3,5,36,37,39,41 are allocated for vRouter, which maps to:
b0000 0010 1011 0000 0000 0000 0000 0000 0000 0000 0010 1011 in binary.

This translates to a hex mask 0x2B00000002B that must be configured into **DPDK_CTRL_TREAD_MASK** variable in /etc/sysconfig/network-scripts/ifcfg-vhost0 file:
DPDK_CTRL_THREAD_MASK=**`0x2B00000002B`**

After file has been updated, the vrouter vhost0 interface must be restarted to take changes into account:

```
$ ifdown vhost0
$ ifup vhost0
```

We can get all contrail-vrouter-dpdk assignments using taskset with -a option :

```
# taskset -cap `pidof contrail-vrouter-dpdk`
pid 21666's current affinity list : 0-71
pid 21740's current affinity list : 0,1,3,5,36,37,39,41
pid 21741's current affinity list : 0,1,3,5,36,37,39,41
pid 21790's current affinity list : 0,1,3,5,36,37,39,41
pid 21791's current affinity list : 10,46
pid 21792's current affinity list : 10,46
pid 21793's current affinity list : 10,46
pid 21794's current affinity list : 10,46
pid 21795's current affinity list : 2
pid 21796's current affinity list : 4
pid 21797's current affinity list : 6
pid 21798's current affinity list : 8
pid 21799's current affinity list : 38
pid 21800's current affinity list : 40
pid 21801's current affinity list : 42
pid 21802's current affinity list : 44
pid 29401's current affinity list : 10,46
```

In the previous example, we can notice that threads are assigned to a given CPU set according following rules :

- **processing threads** have a CPU affinity restricted to a single one CPU (these are 2,4,6,8,38,40,42 and 44). Consequently each is placed on a single CPU.
- **service threads** have CPU affinity restricted to only 2 CPUs (they are 10,46).
- **control threads** have a CPU affinity restricted to the same CPUs assigned for Operation System (they are 0,1,3,5,36,37,39,41).

CPUs assigned to vrouter - Provisioning concerns

Currently **only CPU assigned to packet processing threads** (**CPU_LIST / COREMASK**) can be managed with provisioning tools. Since version 20.03, we can also manage CPU assigned to control and service threads (**DPDK_CTRL_THREAD_MASK** and **SERVICE_CORE_MASK**) with provisioning tools.

Contrail can be installed through different tools: Contrail Cloud, Contrail Networking integrated with RHOSP and TripleO deployer or Juniper Contrail Ansible Deployer to name some.

CPUs assigned to control and service threads (control and service cores) - 20.03 and later release

When setting up the environment we will have to specify the CPU assignment for service threads. For RSHOP and Ansible deployer installation be sure to include the service mask within quotes within configuration files.

When deploying via **Ansible deployer**, file instances.yaml has to include SERVICE_CORE_MASK and DPDK_CTRL_THREAD_MASK variables with quotes in order to avoid the given value to be converted into a decimal number leading to wrong core assignment. So, we have to use SERVICE_CORE_MASK=”0x400000000400” instead of SERVICE_CORE_MASK=0x400000000400 and DPDK_CTRL_THREAD_MASK=”0x2B00000002B” instead of SERVICE_CORE_MASK=0x2B00000002B

```
compute_dpdk_1:
    provider: bms
    ip: 172.30.200.46
    roles:
    vrouter:
    PHYSICAL_INTERFACE: vlan0200
    AGENT_MODE: dpdk
    CPU_CORE_MASK: ”0x154000000154”
    SERVICE_CORE_MASK: ”0x400000000400”
    DPDK_CTRL_THREAD_MASK: ”0x2B00000002B”
    DPDK_UIO_DRIVER: uio_pci_generic
    HUGE_PAGES: 60000
    …
```

When **TripleO** is used, SERVICE_CORE_MASK and DPDK_CTRL_THREAD_MASK variables have to be included into Contrail Service configuration file (*<tripleo-root-dir>*/environments/contrail/contrail-services.yaml). Quotes have also to be used into TripleO Yaml files in order to avoid the value to be converted into decimal.

In TripleO SERVICE_CORE_MASK: ”0x400000000400” has to be used to get vrouter service threads assigned onto 10 and 46 CPU cores, and DPDK_CTRL_THREAD_MASK=”0x2B00000002B” has to be used to get vrouter control threads assigned onto 0,1,3,5,36,37,39 and 41 CPU cores.

SERVICE_CORE_MASK and DPDK_CTRL_THREAD_MASK have to be defined into ContrailSettings.

ContrailSettings variable can be set at role level :

```
parameter_defaults:
  <Role Name>Parameters:
    TunedProfileName: "cpu-partitioning"
    IsolCpusList: "2-35,38-71"
    NovaVcpuPinSet: "12-35,48-71"
    NovaComputeCpuSharedSet: "5-7"
    KernelArgs: " isolcpus=2-35,38-71"
    ContrailSettings:
          SERVICE_CORE_MASK: ”0x400000000400”
          DPDK_CTRL_THREAD_MASK: ”0x2B00000002B”
```

By default, ContrailDpdk is the role name used for DPDK compute.

Or, globally for all roles:

```
parameter_defaults:
    ContrailSettings:
          SERVICE_CORE_MASK: ”0x400000000400”
          DPDK_CTRL_THREAD_MASK: ”0x2B00000002B”
```

/!\: If ContrailSettings is defined at role level, it will hide all ContrailSettings values defined globally. Each ContrailSettings parameters expected at role level have to be redefined even if defined at the global level.

CPUs assigned to packet processing threads (forwarding cores)

When setting up the environment we will have to specify the CPU assignment for packet processing threads. For RSHOP and Ansible deployer installation be sure to include the core mask within quotes within configuration files.

When deploying via **Ansible deployer**, file instances.yaml has to include CPU_CORE_MASK variable with quotes in order to avoid the given value to be converted into a decimal number leading to wrong core assignment. So, we have to use CPU_CORE_MASK=”0x154000000154” instead of CPU_CORE_MASK=0x154000000154.

Here is an example when installing Contrail via Ansible deployer:

```
compute_dpdk_1:
     provider: bms
     ip: 172.30.200.46
     roles:
     vrouter:
     PHYSICAL_INTERFACE: vlan0200
     AGENT_MODE: dpdk
     CPU_CORE_MASK: ”0x154000000154”
     SERVICE_CORE_MASK: ”0x400000000400”
     DPDK_UIO_DRIVER: uio_pci_generic
     HUGE_PAGES: 60000
     …
```

You can connect onto a DPDK compute node and read the vrouter environment file (which is on the host OS, not inside a container) in order to check that the correct value has been configured by the deployer.
If you see something like this, things are correct:

```
[root@server-5b ~]# cat /etc/contrail/common_vrouter.env | grep CORE
CPU_CORE_MASK=0x154000000154
```

Otherwise, if you see this configuration will not be correct:

```
[root@server-5b ~]# cat /etc/contrail/common_vrouter.env | grep CORE
CPU_CORE_MASK=23364622090580
```

Value 23364622090580 is `0x154000000154` in decimal format.

If you find yourself in the wrong scenario then you can simply edit the common_vrouter file so to include the right core mask syntax. Next, “re-create” containers by running:

```
cd to /etc/contrail/vrouter
docker-compose down
docker-compose  up -d
```

This time DPDK will be pinned as desired.

When **TripleO** is used, CPU_LIST has to be included into the Contrail NIC template configuration file (*<tripleo-root-dir>*/network/config/contrail/contrail-dpdk-nic-config.yaml}. Quotes have also to be used into TripleO Yaml files in order to avoid the value to be converted into decimal. In TripleO CPU_LIST="0x154000000154" have to be used to get vrouter processing threads assigned onto 2.4.6,8,38,40,42 and 44 CPU cores.

Here is a Contrail NIC template configuration example when installing Contrail via TripleO deployer:

```
resources:
  OsNetConfigImpl:
    type: OS::Heat::SoftwareConfig
    properties:
     …
          params:
            $network_config:
              network_config:

              - type: contrail_vrouter_dpdk
                name: vhost0
                use_dhcp: false
                cpu_list: ”0x154000000154”
     …
```

NOTE: If more that 54 forwarding cores are selected with CPU_CORE_MASK (or cpu_list), vrouter startup will fail with following message into contrrail-vrouter-dpdk.log file: *"Error configuring lcores: number of forwarding cores exceeds maximum of 54"* .

CPU pinning and isolation

CPU isolation configuration (tuned and isolcpus) - RedHat

First we are enforcing CPU isolation rules in order to keep CPUs assigned to vRouter and Nova out of the system scheduler :

1. Tuned: Prevent CPUs for Nova and vRouter in /etc/tuned/cpu-partitioning-variables.conf

```
cat /etc/tuned/cpu-partitioning-variables.conf
isolated_cores=2-35,38-71
```

2. Isolcpus: Edit grub configuration in order to add isolcpus (and edit grub to hugepages)

```
cat /etc/default/grub
GRUB_CMDLINE_LINUX="console=tty0 console=ttyS0,115200n8 crashkernel=auto rhgb
quiet default_hugepagesz=1GB hugepagesz=1G hugepages=128 hugepagesz=2M
hugepages=40 isolcpus=2-35,38-71
```

3. apply isolation changes :
# systemctl restart tuned
# grub2-mkconfig -o /boot/grub2/grub.cfg
# reboot

CPU isolation configuration (isolcpus) - Ubuntu

On Ubuntu OS, isolcpus mechanism is used in order to keep CPUs assigned to vRouter and Nova out of the system scheduler :

```
Kernel parameter isolcpus=2-35,38-71 in /etc/default/grub, run update-grub and
restart the node.
```

Operating system scheduler CPU assignment (CPUAffinity) - RedHat

1. CPUs assigned to the system scheduler are defined using tuned partitioning mechanism :

```
vi /systemd/system.conf
...
CPUAffinity=0-1,36-37
```

2. then we apply tuned configuration changes :
# systemctl restart tuned

Nova CPU assignment (vcpu_pin_set)

Cores to be used by Nova to run VMs is defined into *vcpu_pin_set* variable (in DEFAULT section in /etc/nova/nova.conf file).

1. We are enforcing Nova CPU assignment :

```
# openstack-config --set /etc/nova/nova.conf DEFAULT vcpu_pin_set 4-47

# cat /etc/nova/nova.conf | grep  vcpu_pin_set
vcpu_pin_set=3,5,7,9,11-35,39,41,43,45,47-71
```

2. In order to get these changes taken into consideration, nova compute service has to be restarted:

```
When OpenStack Kolla is used:
# docker restart nova_compute

When OpenStack Kolla is not used:
# service nova-compute restart (Ununtu)
# service openstack-nova-compute restart (RHEL)
```

PS: Optionally, property *hw:cpu_policy=dedicated* can be set in any openstack flavor definition to disable VCPUs migration from one physical core to another. The advantage of this approach is that for flavors without the ***hw:cpu_policy=dedicated*** property (the default value will be set ***hw:cpu_policy=prefered***), VMs will be balanced between cpus. With *isolcpus* set, all VMs without the above property will end up executing on the first cpu from the list.

vRouter CPU assignment (CPU_LIST, SERVICE_CORE_MASK and DPDK_CTRL_THREAD_MASK)

1. Configure vrouter CPU_LIST (only CPU 2,4,6,8 are used: mask=0x154):

```
 vi /etc/sysconfig/network-scripts/ifcfg-vhost0
[...]  CPU_LIST=0x154
```

PS: CPU 38,40,42 and 44 are not selected and won’t receive any task to process.

2. Configure vrouter SERVICE_CORE_MASK:

```
 vi /etc/sysconfig/network-scripts/ifcfg-vhost0
[...]  SERVICE_CORE_MASK=0x400000000400
```

3. Configure vrouter DPDK_CTRL_THREAD_MASK:

```
 vi /etc/sysconfig/network-scripts/ifcfg-vhost0
[...]  DPDK_CTRL_THREAD_MASK=0x2B00000002B
```

4. We apply vrouter configuration changes :

```
# ifdown vhost0
# ifup vhost0
```

Check CPU pinning on all processes

In order to check actual CPU used to run a process and the CPU pinning rule bound to each process, following command can be used :

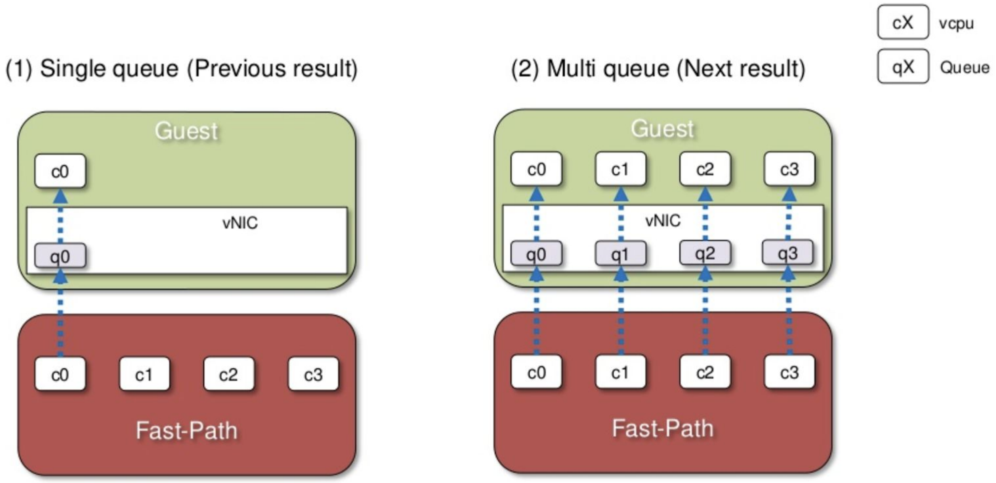
```
$ ps -eT  -o psr,tid,comm,pid,ppid,cmd,pcpu,stat  | awk '$3 != "ps" { cmd="taskset -cp " $2 "| sed \"s/^pid .*s current affinity list:/\" | tr -d \" \" | tr -d \"\\n\" "; cmd | getline affin;  printf( "%6s %s \n", affin, $0)  }' 2> /dev/null | more
       PSR     TID COMMAND             PID    PPID CMD                          %CPU STAT
  0-47   0       1 systemd               1       0 /usr/lib/systemd/systemd --   0.0 Ss
  0-47   0       2 kthreadd              2       0 [kthreadd]                    0.0 S
     0   0       4 kworker/0:0H          4       2 [kworker/0:0H]                0.0 S<
     0   0       6 ksoftirqd/0           6       2 [ksoftirqd/0]                 0.0 S
     0   0       7 migration/0           7       2 [migration/0]                 0.0 S
  0-47   8       8 rcu_bh                8       2 [rcu_bh]                      0.0 S
```

In the first column we can read the ***CPU pinning*** put onto the process with taskset command, in the second one (PSR), we can see the ***actual CPU*** used by the process.

vNIC performance

CPU assigned for Multiqueue and virtIO

When multiqueue is used on a Guest VM, the vrouter will only be able to sustain guest vNIC configured with a number of queue below or equal to the number for processing core allocated to the vrouter. In the ideal scenario, each multiqueue vNIC is configured with the same number of queue than the number of CPU allocated to vrouter for packet processing.



Single queue virtIO

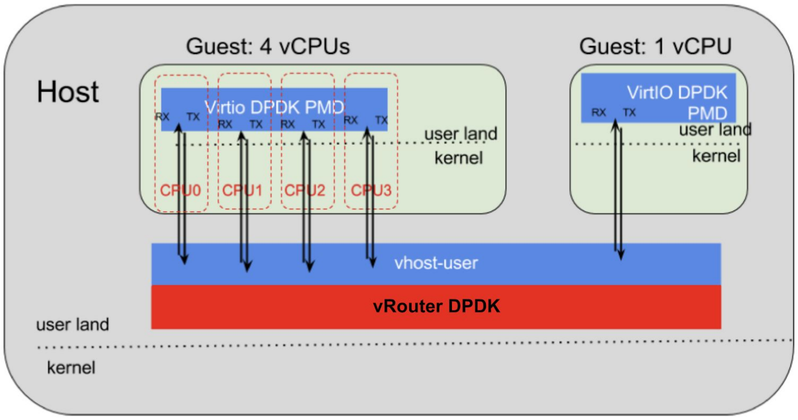
The network performance does not scale as the number of vCPUs increases.
Guest cannot transmit or retrieve packets in parallel as virtio-net have only one TX and RX. virtio-net drivers must be synchronized before sending and receiving packets.

It is useless to assign more CPU to vrouter in order to increase packet processing rate, as only one vrouter CPU is used to poll the single vNIC queue. Moreover, all vrouter CPU could have a packet to transmit to the vNIC single queue which is becoming a congestion point. If we add more CPU, we increase the pressure on the vNIC single queue and it could lead to jitter and even packet drop (buffer overflow).

Multiqueue virtIO

In the case of DPDK vrouter, the support for multiqueue virtio is available from Contrail 3.1 onwards.
The difference with non-DPDK is that vRouter runs in the user space instead of host kernel. Also there is a vhost userspace process wherein the vRouter copies packets to/from the VM.

Every queue inside the VM maps to a virtio ring on the host user space.



Some of the offload capabilities, e.g. GRO (Generic Receive Offload), is not supported with multiqueue virtio on DPDK vrouter in Contrail 3.2.

DPDK Multiqueue packet walk-through

In this case vRouter runs as a multi-core process that exists on all assigned cores. There is also a scale-out approach to packet processing using multiple cores so the performance of one VM is NOT limited by the performance of 1 core.

A vRouter might have more cores than there are queues in the VM. Queues can only be shared when the vRouter cores send to the VM queue. When receiving from the queue, exactly one vRouter core will read from a queue (***i.e. receive queues are not shared, in order to prevent packet reordering***).

vNIC queues queue size configuration (RedHat OpenStack)

vNIC queues are not configured by Contrail vRouter, they are managed by OpenStack.
Rqueue Size are defined at OpenStack level (using libvirt) : /etc/nova/nova.conf

```
# Configure virtio rx queue size.
#
# This option is only usable for virtio-net device with vhost and
# vhost-user backend. Available only with QEMU/KVM. Requires libvirt
# v2.3 QEMU v2.7. (integer value)
# Possible values:
# 256 - <No description provided>
# 512 - <No description provided>
# 1024 - <No description provided>
#rx_queue_size=<None>

# Configure virtio tx queue size.
#
# This option is only usable for virtio-net device with vhost-user
# backend. Available only with QEMU/KVM. Requires libvirt v3.7 QEMU
# v2.10. (integer value)
# Possible values:
# 256 - <No description provided>
# 512 - <No description provided>
# 1024 - <No description provided>
#tx_queue_size=<None>
```

Nova configuration file values are provisioned with TripleO. vNIC queues size can be defined using *NovaLibvirtRxQueueSize* and *NovaLibvirtRxQueueSize* environment parameters. By default they are configured with a 512 value.

Nova configuration file is configured with *<tripleo templates root dir>*/puppet/services/nova-compute.yaml file:

```
 NovaLibvirtRxQueueSize:
    description: >
      virtio-net RX queue size. Valid values are 256, 512, 1024
    default: 512
    type: number
    constraints:
      - allowed_values: [ 256, 512, 1024 ]
    tags:
      - role_specific
  NovaLibvirtTxQueueSize:
    description: >
      virtio-net TX queue size. Valid values are 256, 512, 1024
    default: 512
    type: number
    constraints:
      - allowed_values: [ 256, 512, 1024 ]
```

Using virsh dumpxml command, we can verify queue size configured by Nova with libvirt :

```
virsh dumpxml [vm id]
...
<interface type='ethernet'>
   <model type='virtio'/>
   <driver name='vhost' queues='5' rx_queue_size='1024' tx_queue_size='1024'/>
</interface>
...
```

multiQ vNIC and number of queues size configured (RedHat OpenStack)

Nova is configuring automatically the number of queues of each vNIC. There is no way to manually define the number of queues to be created for each vNIC. We can only change the number of queues of a given vNIC after the virtual instance has been started, using some Linux commande into the virtual machine whose vNIC configuration has to be modified (cf next section).

Nova is using vif.py script to create instance virtual interfaces
(https://github.com/openstack/nova/blob/master/nova/virt/libvirt/vif.py)
Number of configured queues for a vif is depending on:

- Compute node kernel release
- Number of vCPU configured onto the virtual instance

When a Linux Kernel 3.x is used (RHOSP 13), a maximum of 8 queues are configured by Nova for each vif.

- when an instance is configured with 8 or less vCPUs: the number of queues configured on its multiqueues vif is equal to the number of vCPU.
- when an instance is configured with more than 8 vCPUs: the number of queues configured on its multiqueues vif is equal 8

When a Linux Kernel 4.x is used (RHOSP 16), the same algorithm is used but using 256 instead of 8 as a maximum of queues to be configured by each vif.

```
$ vi /usr/lib/python2.7/site-packages/nova/virt/libvirt/vif.py
...
    def _get_max_tap_queues(self):
        # NOTE(kengo.sakai): In kernels prior to 3.0,
        # multiple queues on a tap interface is not supported.
        # In kernels 3.x, the number of queues on a tap interface
        # is limited to 8. From 4.0, the number is 256.
        # See: https://bugs.launchpad.net/nova/+bug/1570631
        kernel_version = int(os.uname()[2].split(".")[0])
        if kernel_version <= 2:
            return 1
        elif kernel_version == 3:
            return 8
        elif kernel_version == 4:
            return 256
        else:
            return None
...
```

Enable Multiqueue on OpenStack instances

To enable multiqueue with DPDK in a Glance image, we add metadata hw_vif_multiqueue_enabled="true":

```
openstack image set --property hw_vif_multiqueue_enabled="true" [image name or ID]
```

Number of queues to be created can’t be specified with Glance metadata. Currently, the number of queues will match the number of vCPUs, defined for the instance.

Using virsh dumpxml command, we can verify how many queues have been created by libvirt :

```
virsh dumpxml [vm id]
...
<interface type='ethernet'>
   <model type='virtio'/>
   <driver name='vhost' queues='5'/>
</interface>
...
```

Here, 5 queues have been allocated to the vNIC.

How to change the number of queues for an OpenStack instances vNIC

Inside an guest instantiated guest with multiqueues enabled, the NIC channel setup can be checked and changed as needed with the commands below.

In order to see the current number of queues we have to use :

```
ethtool -l [nic name]
```

In order to change the number of queues we have to use :

```
ethtool -L [nic name] combined [number of queues]
```

For instance, below we are configuring 4 logical queues on eth0 vNIC :

```
ethtool -L eth0 combined 4
```

This should be done during interface initialization, for example in a “pre-up” action in /etc/network/interfaces. But it seems to be possible to change this configuration on a running interface without disruption.

PS: above commands have to be issued **inside each virtual machine**.

Multiqueue and known limitations

Number of queues on VM instances:

Number of queues configured on virtual instances collected by the vrouter **must be below or equal to the number of queues configured on the vrouter** (aka number of CPU assigned to vrouter for packet processing).

If more queues are configured on the instance side than on the vrouter side, following messages will be seen into the vrouter dpdk logs :

```
2019-09-24 16:36:50,011 VROUTER: Adding vif 8 (gen. 37) virtual device tap66e68bc1-a9
2019-09-24 16:36:50,012 VROUTER:     lcore 12 TX to HW queue 0
2019-09-24 16:36:50,012 VROUTER:     lcore 13 TX to HW queue 1
2019-09-24 16:36:50,012 VROUTER:     lcore  8 TX to HW queue 2
2019-09-24 16:36:50,012 VROUTER:     lcore  9 TX to HW queue 3
2019-09-24 16:36:50,012 VROUTER:     lcore 10 TX to HW queue 4
2019-09-24 16:36:50,012 VROUTER:     lcore 11 TX to HW queue 5
2019-09-24 16:36:50,012 VROUTER:     lcore 12 RX from HW queue 0
2019-09-24 16:36:50,012 VROUTER:     lcore 13 RX from HW queue 1
2019-09-24 16:36:50,012 VROUTER:     lcore 10 RX from HW queue 2
2019-09-24 16:36:50,012 VROUTER:     lcore 11 RX from HW queue 3
2019-09-24 16:36:50,012 UVHOST: Adding vif 8 virtual device tap66e68bc1-a9
2019-09-24 16:36:50,012 UVHOST:     vif (server) 8 socket tap66e68bc1-a9 FD is 173
….
2019-09-24 16:37:46,692 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 0 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 1 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 2 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 3 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 4 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 5 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 6 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 7 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: vr_uvhm_set_vring_enable: Can not disable TX queue 4 (only 4 queues)
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
```

```
2019-09-24 16:37:46,693 UVHOST: vr_uvhm_set_vring_enable: Can not disable RX queue 4 (only 4 queues)
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: vr_uvhm_set_vring_enable: Can not disable TX queue 5 (only 4 queues)
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: vr_uvhm_set_vring_enable: Can not disable RX queue 5 (only 4 queues)
….
019-09-24 16:40:58,250 UVHOST: vr_uvhm_set_vring_enable: Can not enable TX queue 7 (only 4 queues)
2019-09-24 16:40:58,250 UVHOST: Client _tap3c0c41f1-66: handling message 18
2019-09-24 16:40:58,250 UVHOST: vr_uvhm_set_vring_enable: Can not enable RX queue 7 (only 4 queues)
```

Here we have a vrouter configured with 4 forwarding cores (= 4 queues) but the collected VM is configured with 8 queues. Queues 4 to 7 won’t be processed by the vrouter.

Indirect descriptors are not supported:

Contrail does not support indirect buffer descriptors (VIRTIO_RING_F_INDIRECT_DESC).
Cf: https://spp-tmp.readthedocs.io/en/stable/rel_notes/release_16_11.html
So, indirect_desc field is set to false in QEMU interface setup :

```
[heat-admin@xxxx ~]$ sudo docker exec -it c149193e4a14 virsh qemu-monitor-command 1 --hmp info qtree
bus: main-system-bus
  type System
  dev: kvm-ioapic, id ""
    gpio-in "" 24
    gsi_base = 0 (0x0)
    mmio 00000000fec00000/0000000000001000
  dev: i440FX-pcihost, id ""
    pci-hole64-size = 2147483648 (2 GiB)
    short_root_bus = 0 (0x0)
    x-pci-hole64-fix = true
    bus: pci.0
      type PCI
...
      dev: virtio-net-pci, id "net1"
        ioeventfd = true
        vectors = 18 (0x12)
        virtio-pci-bus-master-bug-migration = false
...
        bar 1: mem at 0xfebd2000 [0xfebd2fff]
        bar 6: mem at 0xffffffffffffffff [0x3fffe]
        bus: virtio-bus
          type virtio-pci-bus
          dev: virtio-net-device, id ""
            csum = true
…
            rx_queue_size = 1024 (0x400)
            tx_queue_size = 1024 (0x400)
            host_mtu = 0 (0x0)
            x-mtu-bypass-backend = true
            speed = -1 (0xffffffffffffffff)
            duplex = ""
            indirect_desc = false
            event_idx = false
            notify_on_empty = false
            any_layout = false
            iommu_platform = false
            __com.redhat_rhel6_ctrl_guest_workaround = false
...
```

IRQ setup

Irqbalance

Disabling IRQ Balance is not recommended as it has no impact on performance. In RHEL, tuned's cpu-partitioning profile leverages IRQ CPU affinity (`isolated_cores`).

Disable IRQ Balance

First, stop irqbalance daemon :
# systemctl stop irqbalance

Then, remove irqbalance from system startup :
# systemctl disable irqbalance

Exclude CPUs from IRQ Balancing

The /etc/sysconfig/irqbalance configuration file contains a parameter named *IRQBALANCE_BANNED_CPUS* that allows to define CPUs to be excluded from consideration by the IRQ balancing service.

The following mask excludes CPUs 8 to 15 as well as CPU 33 from IRQ balancing:

IRQBALANCE_BANNED_CPUS=00000001,0000ff00

PS: If you are running a system with up to 64 CPU cores, separate each group of eight hexadecimal digits with a comma.

Unsupported features

1. QoS is not supported (Marking is supported with DPDK vrouter. Hardware queueing is not).
2. Jumbo frames for DPDK VMs in 3.2.8 and 4.0.1 (*).
3. Jumbo frames for non-DPDK VMs on DPDK nodes in 3.2.8 and 4.0.1 (*)

(*) these features are requiring mergeable buffers to be supported.

Deployment considerations

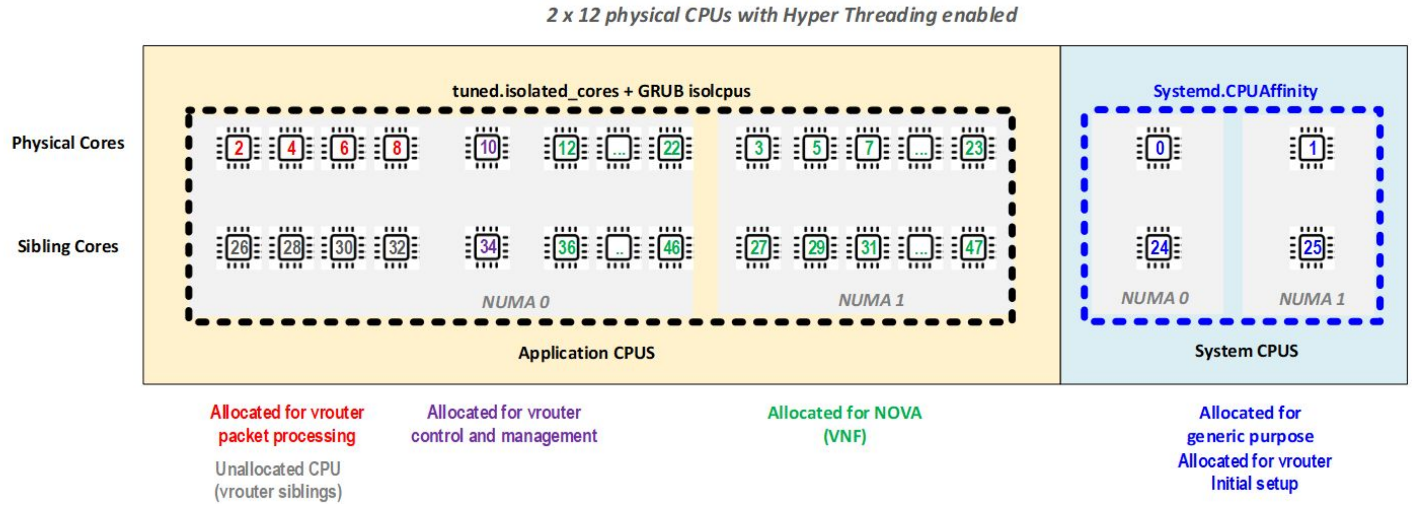
Define number of resources required for vRouter DPDK

DPDK vrouter dimensioning is depending on following parameters:

- Cores for vrouter forwarding
- Hugepages for vrouter
- Cores for vrouter services threads and DPDK control threads

Compute Node CPU capacity planning

When we are designing an OpenStack Contrail infrastructure we have to define how many CPU will be used for vRouter needs, how many for virtual instances and how many will remain to be used by the Operating system.

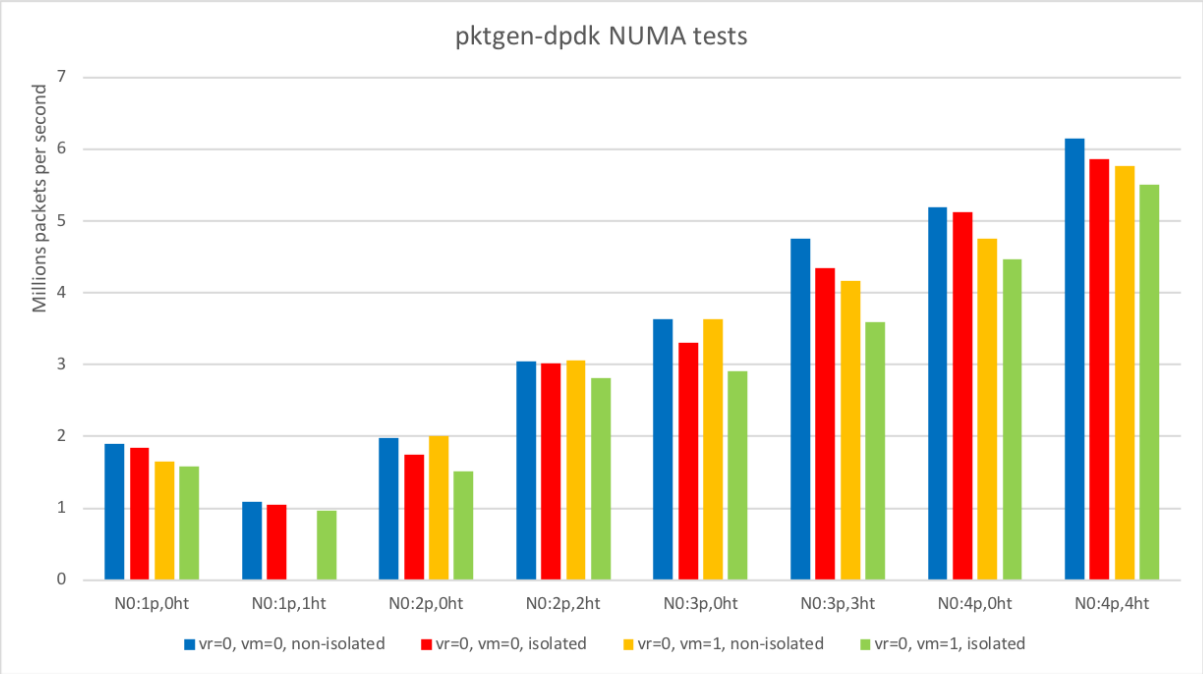

2 x 12 physical CPUs with Hyper Threading enabled
Physical Cores
Sibling Cores
tuned.isolated_cores + GRUB isolcpus
2
4
6
8
10
12
...
22
3
5
7
...
23
26
28
30
32
34
36
..
46
27
29
31
...
47
NUMA 0
NUMA 1
Application CPUS
Systemd.CPUAffinity
0
1
24
25
NUMA 0
NUMA 1
System CPUS
Allocated for vrouter packet processing
Unallocated CPU (vrouter siblings)
Allocated for vrouter control and management
Allocated for NOVA (VNF)
Allocated for generic purpose
Allocated for vrouter Initial setup

Performance tests figures (DPDK vrouter)

In best case scenarios the vRouter can achieve 1,5Mpps per physical core. The actual performance of the system depends on several parameters: VNF capabilities, traffic patterns, CPU, RAM speed, NICs capabilities, drivers, firmware, etc. Juniper Networks recommends performing tests using a lab representative of the target production environment.

For standard deployments, 4 up to 8 physical cores are allocated. When HT siblings are not assigned for vRouter they **MUST NOT** be assigned for Nova or operating system because this can cause CPU steals.

In the next diagram we can see that with a single lcore allocated to the vrouter, we've got around 1.8 MPPS performance. When 2 lcores are allocated to the vrouter, performance is falling to 0.5 MPPS per core (this is the impact of internal load balancing which is very inefficient for only 2 lcores). When 4 or more lcores are allocated to the vrouter, we can see a performance of 0.8 MPPS per lcore when siblings are used (1.6 MPPS per physical CPUl) and a performance of 1.2 MPPS per lecore when siblings are not used (1.2 MPPS at per physical CPU).



Juniper QA tests - 2018

<u>Diagram legend</u>:

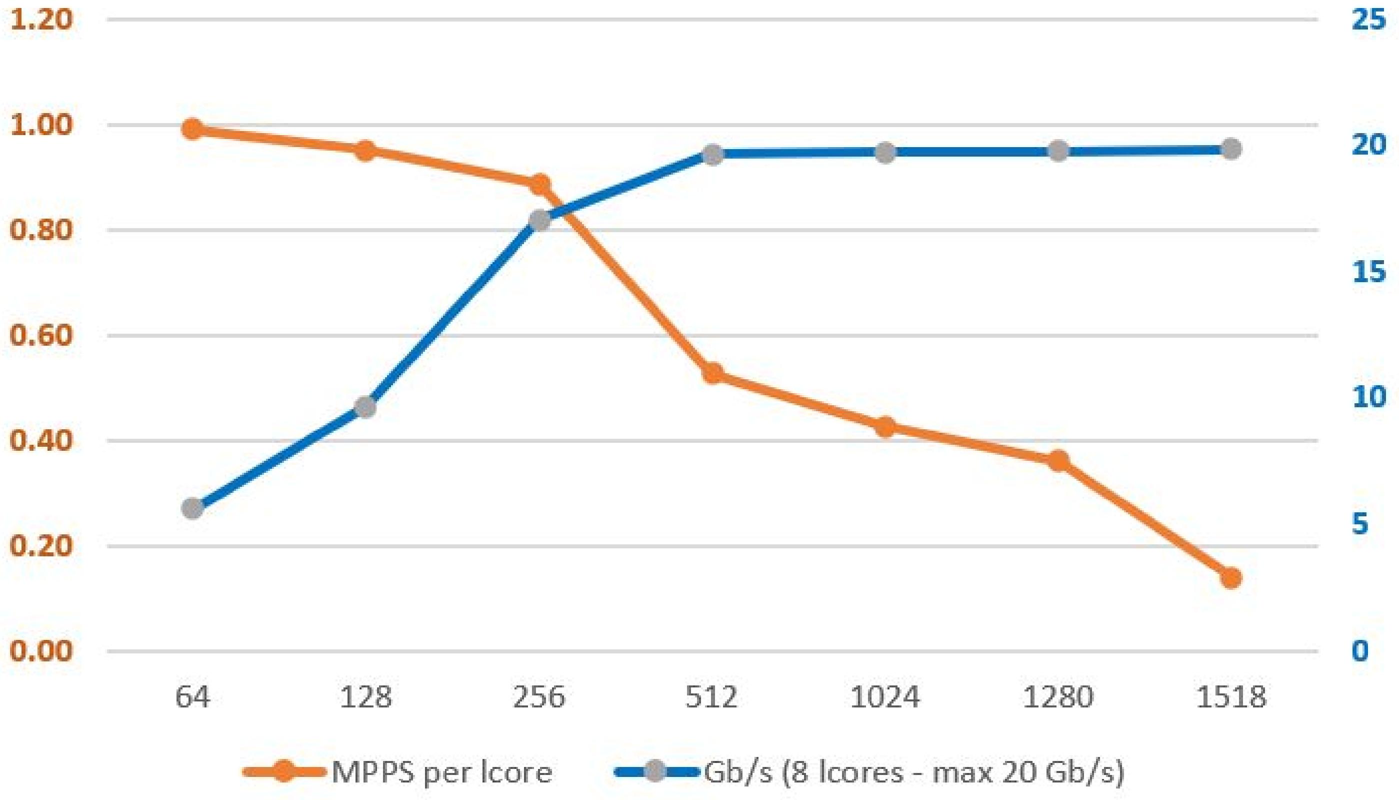
- vr - vRouter allocate on NUMA 0 or 1
- vm - VM allocated on NUMA 0 or 1
- [X]p[Y]ht - number of allocated physical cores with HT siblings

We can summarize this diagram performance result into following table:

	With Siblings		**Without Siblings**	
	per lcore	per Phys. CPU	per lcore	per Phys. CPU
1 lcore	1.8 MPPS	1.8 MPPS	1.8 MPPS	1.8 MPPS
2 lcores	0.5 MPPS	1.0 MPPS	1.0 MPPS	1.0 MPPS
>= 4 lcores	0.75 MPPS	1.5 MPPS	1.25 MPPS	1.25 MPPS

These tests have been done without DPDK control threads and vrouter service threads pinning.
So, we can also see that before release 20.03 (or without the patch delivered for 19.12 release) it is preferable not to enforce any CPU isolation rules to get the best performance as it has an impact on packet drops. So in release 19.12 and earlier (without patch applied), for a given level of packet drops, performance without any CPU isolation rule is better.

In 2020, new tests have been proceeded (in Juniper and customer labs) with DPDK control threads vrouter service threads pinning. In these tests (done on RedHat OpeStack platform + contrail 19.12 + patch for DPDK control threads and vrouter service threads pinning) we've enforced CPU isolation (with isolcpus and tuned).



Juniper and customer tests summary - 2020 (19.12 + patch) - siblings used

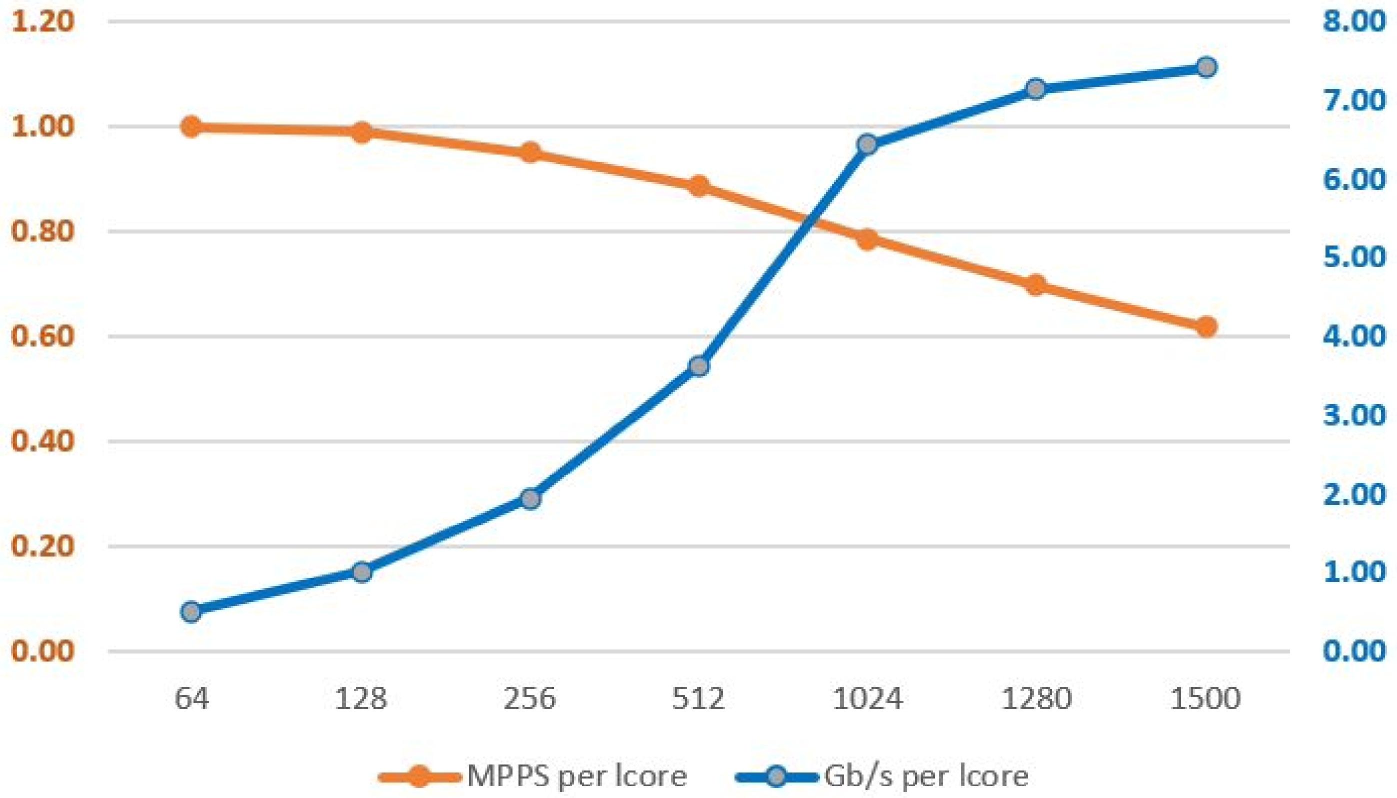
These tests have been done with 8 lcores and CPU siblings were used. vRouter physical interface was a bond of 2x10 Gb/s NIC (Intel Niantic cards).

MPPS per lcore is the number of packet per second that are processed at vrouter level. Given values are the performance seen at user level for a half duplex flow (outgoing only). For a full duplex flows (both outgoing and incoming direction) these values have to be divided by two.

On such a platform when packet size is more than 512 bytes we were reaching the physical interface maximum throughput. This is why performance per lcore seems to decrease at a very low rate for higher packet size.

We can use the results we've got for packet size below 512 bytes to extrapolate values for a higher packet size when we have more than 20 Gb/s available on vrouter physical interface.

In the diagram below, the MPPS per lcore values for 1024, 1280 and 1500 packet have been extrapolated using 64, 128, 256 and 512 packet size values :



This diagram could help our customers to evaluate the number of lcores needed to get a given performance at vrouter level. Of course these values are dependent on the customer's actual setup and have to be revalidated on field. Real results can differ from what we've got in Juniper labs.

The best results will be achieved when:

- vRouter, VNF and NICs will be on the same NUMA. It does not mean that other configurations are not allowed but some performance degradation can be observed (depends on traffic characteristic and NUMA allocation).
- Contrail vrouter threads are correctly pinned (available since 20.03 version) :
  - using CPU_LIST for packet processing threads

    - using SERVICE_CORE_MASK and DPDK_CTRL_THREAD_MASK for control and service threads (only available since 20.03 version)
    - core siblings are not used for physical core allocated to packet processing threads (cf: vRouter CPU capacity planning section)
- CPU allocated to DPDK vrouter and Nova are properly isolated with both *tuned partitioning* and *isolcpus* mechanisms (contrail 20.03 and later version).

/!: When using a Contrail Release 19.12 or earlier, some erratic drops can be seen when CPU isolation rules are enforced either on vrouter and or nova allocated CPUs. For these versions the bests results are reached when no isolation mechanisms are enforced.

vrouter forwarding cores capacity planning

First step: expected network throughput

Expected network throughput is the first thing to take into consideration.
We can take following assumption for vRouter dimensioning (this is the best scenario with an optimal setup) :

(*)	64 bytes packets	512 bytes packets	1500 bytes packets
Icore (sibling enabled)	1 MPPS	0.9 MPPS	0.6 MPPS
physical CPU	2 MPPS	1.8 MPPS	1.2 MPPS

(*): these values are depending on the actual customer environment. Real figures could be different.

Example - Scenario 1:
Let’s assume we want to be able to process 10 Gb/s of 512 bytes packet size in full duplex, we need 5 MPPS throughput at the vrouter level (2 x 10 x 1000 / 512 / 8).
So, we need at least 6 forwarding cores to be configured onto the vRouter (5 / 0.9).

Example - Scenario 2:
We need to be able to process 12 MPPS of 64 bytes packets onto our vRouter (in full duplex mode: 6 MPPS RX + 6 MPPS TX). So we need at least 12 forwarding cores (12 / 1).

Second step: virtual instance DPDK support and multi queues capability

First topic to consider is to determine whether the virtual instances are using or not DPDK. And, if DPDK is supported, do the virtual instances support multiQ vNIC ?

case 1: not DPDK capable or are not supporting DPDK multiQ vNIC:
It is preferable not to use more than 4 forwarding cores onto the vRouter. If more than 4 forwarding cores are used onto the vrouter, some drop can occur at virtIO queue level due to the traffic burst that will be generated by the usage of several forwarding cores to deliver packet into a single Q.

case 2: DPDK capable multiQ vNIC:
The number of vRouter forwarding cores to be used will depend on:

- Linux Kernel release used onto the OpenStack compute node
- Maximum number of vCPU configured on DPDK virtual instance

The table below is providing the vRouter allocation rule to follow:

	Number of Forwarding cores to use	
Max Number of vCPU	Linux Kernel 3.X	Linux Kernel 4.X
N < 8	>= N	>= N
N >= 8	>= 8	>= N

Example - Scenario 1 and 2:
Let’s assume we are using MultiQ vNIC virtual instances on RedHat OpenStack 13 platform. So, Linux Kernel 3.X will be used on compute nodes. These virtual instances will be configured with 12 vCPU. As Linux Kernel 3.X is used onto our compute nodes, each multiQ vNIC will be configured with 8 queues at maximum. So we need to use at least 8 forwarding cores to be configured onto the vRouter.

Last step: customer use case and sibling consideration

Then we have to take into consideration the purpose of the OpenStack Contrail platform.

case 1: high network throughput per vNIC
If the customer is intending to use its OpenStack computes to host virtual instances requiring very high network throughput per vNIC. It is sometimes preferable not to use siblings on forwarding cores. If sibling is not used, we could expect a 50% performance improvement per processing core.

(*)	64 bytes packets	512 bytes packets	1500 bytes packets
lcore (sibling enabled)	1.5 MPPS	1.3 MPPS	0.9 MPPS
physical CPU	1.5 MPPS	1.3 MPPS	0.9 MPPS

(*): these values are depending on the actual customer environment. Real figures could be different.

case 2: as many as possible virtual instances per compute node
If the customer is willing to keep a better capacity on its compute nodes in order to start as many virtual instances as possible, it is preferable to use CPU siblings on forwarding cores.

Example - Scenario 1:
We are using 2 x 12 physical CPU computes. We need 5 Gb/s full duplex traffic throughput to be processed by Contrail vRouter (at least 6 forwarding core required). As we are using MultiQ Virtual instances with more than 8 vCPU onto Compute nodes running Linux 3.x Kernel, we have to allocate at least 8 forwarding cores onto Contrail vRouter.
We do not foresee a higher network throughput expectation than the hypothesis taken for our current vRouter dimensioning. So we are preferring to keep more CPU for VM usage instead of Network performance improvement. We will use our CPU siblings for our forwarding cores.

So, we will use a vRouter configuration with 8 forwarding cores configured onto 4 Physical CPUs on NUMA0. Our compute nodes are fitted with 12 Physical CPUs per NUMA. So, 8 Physical CPUs - 16 logical cores - will remain available for other purposes (7 allocated - 14 logical cores - to VM instances and 1 allocated - 2 logical cores to Linux Operating System for Hypervisor needs).

Example - Scenario 2:
We are using 2 x 12 physical CPU computes. We need 12 MPPS throughput on our vRouter (64 bytes packets) . At least 12 logical cores are required to get such a performance. As we are using MultiQ Virtual instances with more than 8 vCPU onto Compute nodes running Linux 3.x Kernel, we have to allocate at least 8 forwarding cores onto Contrail vRouter.

- If sibling is used we need to use 12 logical cores on our vRouter (6 Physical CPUs)
- If sibling is not used we need 8 physical cores on our vRouter (8 physical CPUs)

If we are using siblings we are saving 2 Physical CPUs. But, we probably won’t be able to reach the expected throughput. Indeed, our multiQ vNIC are configured with 8 queues. If 8 physical cores are used, we will be able to process each vNIC queue at 1.5 MPPS. So 12 MPPS for the eight queues (6 MPPS RX + 6 MPPS TX). But, if we are using siblings, we will only be able to process 1 MPPS each vNIC queue at 1 MPPS. So only 8 MPPS for the eight queues (4 MPPS RX + 4 MPPS TX).

Siblings or not siblings :

- If we need 12 MPPS on vRouter distributed on several vNIC, we could use only 6 physical cores with their siblings (12 logical cores in total).
- If we need 12 MPPS for a single vNIC, it is preferable to use only 8 physical cores. We have a virtual instance which has **high network throughput requirements per vNIC**.

vrouter DPDK control threads and services cores capacity planning

DPDK control threads

DPDK control threads can be pinned on any CPU. These CPUs are only used during Contrail vRouter startup for DPDK initialization. As they did not require high CPU capacity, a good strategy is to use the CPU allocated to the Operating System for these threads.

vRouter service threads

Service threads are used into the communication between vRouter dataplane (DPDK vrouter process) and the control plane (vrouter agent). They are involved in vNIC plugin to the vRouter, flow creation/deletion onto the vrouter. So we have two scenarios to consider :

case 1: vrouter is used only in Packet Mode
One single CPU will be enough for Service Threads. This CPU could be chosen on any NUMA node. 1 CPU on NUMA 1 enough. If hyperthreading is enabled, a single logical core would be enough.

case 2 : vrouter is used Flow Mode
We are allocating a single Physical CPU will be allocated to services threads. If hyperthreading is enabled, both logical cores will be allocated to Service Threads. If lots of flows are required at a high rate, an additional CPU could be needed. These CPUs could be chosen on any NUMA node (but on the same NUMA).

Example:
We are running most of our vNIC in packet mode. Only a few of them will be using flow mode. We will allocate a Physical CPU on NUMA1 for vrouter Service Threads (as our forwarding cores have been allocated on NUMA0). We will use both logical cores on this physical CPU for our Service Threads.

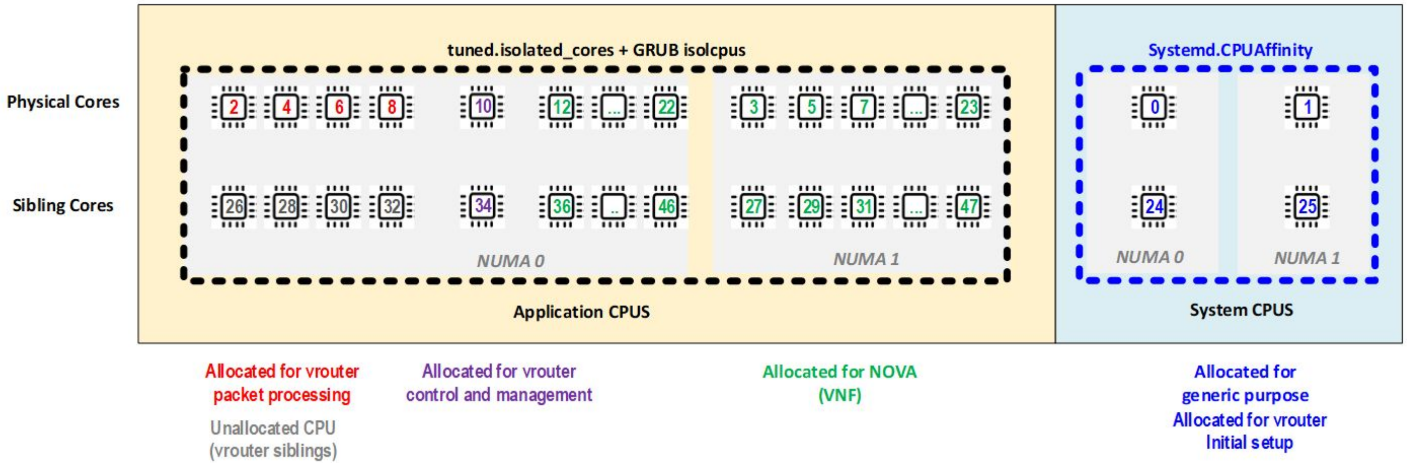
DPDK control threads will be allocated onto Operating System assigned CPUs

CPU assignment typical setups

Onto a Contrail OpenStack platform we have following kind of processes that will use CPUs:

- Operating system processes
- Nova compute etc processes
- vRouter DPDK service lightweight processes
  - packet processing threads
  - control and service threads **(20.03 and later release)**
- vRouter Agent (default is 2, this can be upgraded to 4 in high flow environment)
- vrouter DPDK forwarding thread

2 x 12 physical CPUs with Hyper Threading enabled


tuned.isolated_cores + GRUB isolcpus
Systemd.CPUAffinity
Physical Cores
Sibling Cores
2
4
6
8
10
12
...
22
3
5
7
...
23
0
1
26
28
30
32
34
36
...
46
27
29
31
...
47
24
25
NUMA 0
NUMA 1
NUMA 0
NUMA 1
Application CPUS
System CPUS
Allocated for vrouter packet processing
Unallocated CPU (vrouter siblings)
Allocated for vrouter control and management
Allocated for NOVA (VNF)
Allocated for generic purpose
Allocated for vrouter Initial setup

We are defining 4 CPU groups:

- ❏ CPUs used for DPDK vRouter packet processing (aka packet processing threads)
- ❏ CPUs used for DPDK vRouter control and management (service threads)
- ❏ CPUs used by VNF (aka CPU managed by Nova)
- ❏ CPUs used by the Operating System and any other task not listed above and DPDK vRouter initial setup (DPDK control threads).

We are applying these allocation rules to get the best performance as possible:

- When Hyper Threading is used, we will book both physical CPUs and their sibling ones for DPDK vRouter packet processing threads.
- Only physical CPUs will be used and assigned to vRouter packet processing threads. Sibling CPUs are unused.
- All CPUs assigned to DPDK vRouter packet processing threads must be taken onto the same NUMA.

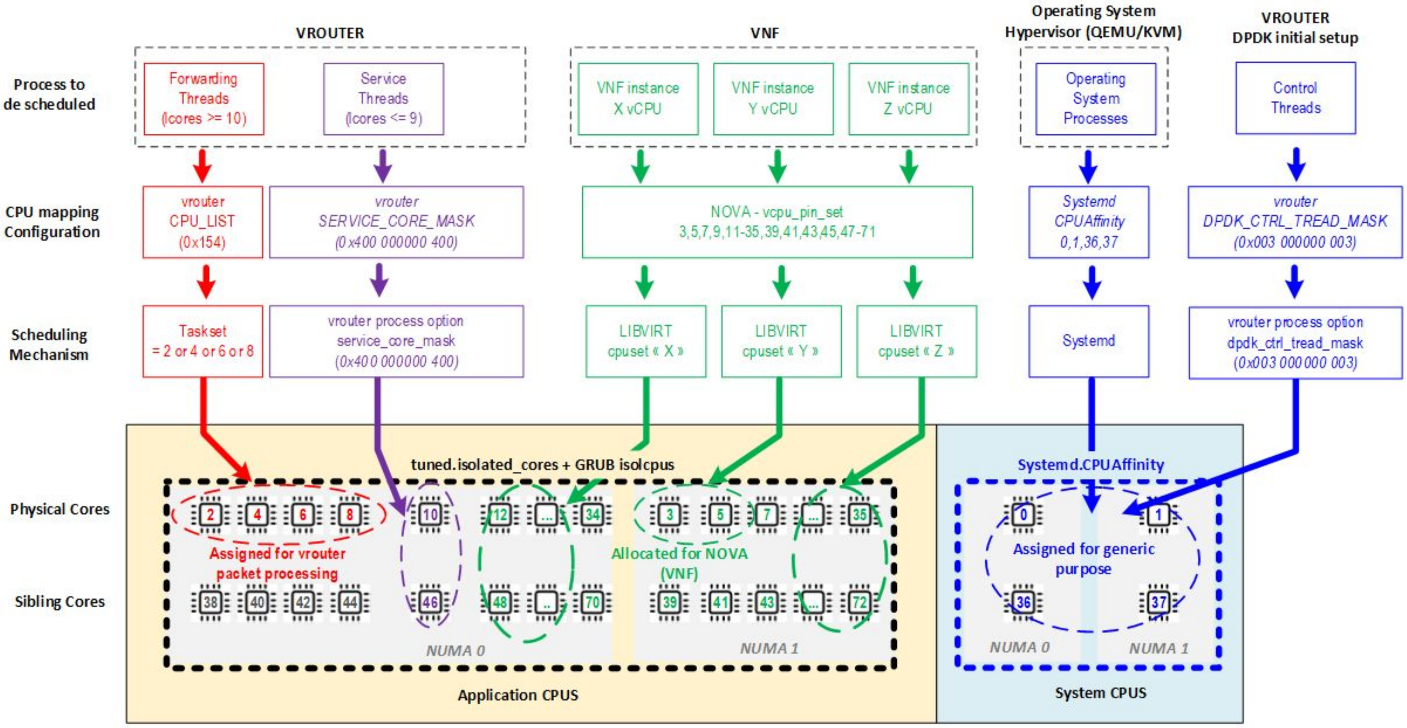
vRouter assigned CPUs and Nova assigned CPU will be isolated from in order to prevent them from being used by any other application (or any Operating System task). Both isolcpu and tuned partitioning mechanisms will be used to get the best isolation as possible.

In the following pages, we will describe a performance tuning configuration using a node with 36 physical cores split on 2 NUMA and Hyperthreading enabled (72 logical cores).

On this system we are applying the following typical CPU allocation:

- CPUs used for DPDK vRouter packet processing: 2,4,6,8,38,40,42,44 (NUMA 0)
  - only 4 of them will be assigned to vrouter: 2,4,6,8
  - remaining ones are kept unassigned: 38,40,42,44
- CPUs used for DPDK vRouter service threads: 10,46 (NUMA 0)
- CPUs assigned to VNF (aka Nova CPU): 3,5,7,9,11-35,39,41,43,45,47-71 (NUMA 0 and 1)
- CPUs assigned to the operating system and other task (vrouter DPDK control threads): 0,1,36,37 (NUMA 0 and 1)

Configuration described into the next pages is really relevant when a Contrail 20.03 and later release is used. CPUs 2-35,38-71 will be allocated to Contrail DPDK vRouter and to Nova. They will be isolated using both ***tuned partitioning*** and ***isolcpus*** mechanisms. The diagram below is describing the target situation:


VROUTER
VNF
Operating System Hypervisor (QEMU/KVM)
VROUTER DPDK initial setup
Process to de scheduled
Forwarding Threads (lcores >= 10)
Service Threads (lcores <= 9)
VNF instance X vCPU
VNF instance Y vCPU
VNF instance Z vCPU
Operating System Processes
Control Threads
CPU mapping Configuration
vrouter CPU_LIST (0x154)
vrouter SERVICE_CORE_MASK (0x400 000000 400)
NOVA - vcpu_pin_set 3,5,7,9,11-35,39,41,43,45,47-71
Systemd CPUAffinity 0,1,36,37
vrouter DPDK_CTRL_TREAD_MASK (0x003 000000 003)
Scheduling Mechanism
Taskset = 2 or 4 or 6 or 8
vrouter process option service_core_mask (0x400 000000 400)
LIBVIRT cpuset « X »
LIBVIRT cpuset « Y »
LIBVIRT cpuset « Z »
Systemd
vrouter process option dpdk_ctrl_tread_mask (0x003 000000 003)
tuned.isolated_cores + GRUB isolcpus
Systemd.CPUAffinity
Physical Cores
Sibling Cores
2
4
6
8
10
12
...
34
3
5
7
...
35
0
1
Assigned for vrouter packet processing
Allocated for NOVA (VNF)
Assigned for generic purpose
38
40
42
44
46
48
...
70
39
41
43
...
72
36
37
NUMA 0
NUMA 1
NUMA 0
NUMA 1
Application CPUS
System CPUS

PS: When using a Contrail release 19.12 and earlier, this is not recommended to enforce any CPU isolation rule (for both vrouter and nova assigned CPU). As these release are not supporting control and service threads CPU pinning, any isolation rule could generate some instabilities and random packet drop period)

Discover hardware architecture topology

Use lstopo and numactl commands to check cores numbering and numa nodes architecture

```
(Ubuntu) # apt-get install hwloc
(RedHat) # yum install hwloc

# lstopo
Machine (252GB)
  NUMANode L#0 (P#0 126GB)
    Socket L#0 + L3 L#0 (30MB)
      L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
        PU L#0 (P#0)
        PU L#1 (P#24)
      L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
        PU L#2 (P#1)
        PU L#3 (P#25)
      L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2
        PU L#4 (P#2)
        PU L#5 (P#26)
      L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3
        PU L#6 (P#3)
        PU L#7 (P#27)
      L2 L#4 (256KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4
        PU L#8 (P#4)
        PU L#9 (P#28)
      L2 L#5 (256KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5
        PU L#10 (P#5)
        PU L#11 (P#29)
      L2 L#6 (256KB) + L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6
        PU L#12 (P#6)
        PU L#13 (P#30)
      L2 L#7 (256KB) + L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7
        PU L#14 (P#7)
        PU L#15 (P#31)
      L2 L#8 (256KB) + L1d L#8 (32KB) + L1i L#8 (32KB) + Core L#8
        PU L#16 (P#8)
        PU L#17 (P#32)
      L2 L#9 (256KB) + L1d L#9 (32KB) + L1i L#9 (32KB) + Core L#9
        PU L#18 (P#9)
        PU L#19 (P#33)
      L2 L#10 (256KB) + L1d L#10 (32KB) + L1i L#10 (32KB) + Core L#10
        PU L#20 (P#10)
        PU L#21 (P#34)
      L2 L#11 (256KB) + L1d L#11 (32KB) + L1i L#11 (32KB) + Core L#11
        PU L#22 (P#11)
        PU L#23 (P#35)
    HostBridge L#0
      PCIBridge
        PCI 103c:3239
          Block L#0 "sda"
          Block L#1 "sdb"
      PCIBridge
        PCI 8086:1572
          Net L#2 "t1a"
        PCI 8086:1572
          Net L#3 "t1b"
      PCIBridge
        PCI 8086:1572
          Net L#4 "t2a"
        PCI 8086:1572
          Net L#5 "t2b"
      PCIBridge
        PCI 102b:0533
      PCIBridge
        PCI 14e4:1657
          Net L#6 "meth0"
        PCI 14e4:1657
          Net L#7 "meth1"
        PCI 14e4:1657
          Net L#8 "eth4"
        PCI 14e4:1657
          Net L#9 "eth6"
  NUMANode L#1 (P#1 126GB) + Socket L#1 + L3 L#1 (30MB)
    L2 L#12 (256KB) + L1d L#12 (32KB) + L1i L#12 (32KB) + Core L#12
      PU L#24 (P#12)
      PU L#25 (P#36)
    L2 L#13 (256KB) + L1d L#13 (32KB) + L1i L#13 (32KB) + Core L#13
      PU L#26 (P#13)
      PU L#27 (P#37)
    L2 L#14 (256KB) + L1d L#14 (32KB) + L1i L#14 (32KB) + Core L#14
      PU L#28 (P#14)
      PU L#29 (P#38)
    L2 L#15 (256KB) + L1d L#15 (32KB) + L1i L#15 (32KB) + Core L#15
      PU L#30 (P#15)
      PU L#31 (P#39)
    L2 L#16 (256KB) + L1d L#16 (32KB) + L1i L#16 (32KB) + Core L#16
      PU L#32 (P#16)
      PU L#33 (P#40)
    L2 L#17 (256KB) + L1d L#17 (32KB) + L1i L#17 (32KB) + Core L#17
      PU L#34 (P#17)
      PU L#35 (P#41)
    L2 L#18 (256KB) + L1d L#18 (32KB) + L1i L#18 (32KB) + Core L#18
      PU L#36 (P#18)
      PU L#37 (P#42)
    L2 L#19 (256KB) + L1d L#19 (32KB) + L1i L#19 (32KB) + Core L#19
      PU L#38 (P#19)
      PU L#39 (P#43)
    L2 L#20 (256KB) + L1d L#20 (32KB) + L1i L#20 (32KB) + Core L#20
      PU L#40 (P#20)
      PU L#41 (P#44)
    L2 L#21 (256KB) + L1d L#21 (32KB) + L1i L#21 (32KB) + Core L#21
      PU L#42 (P#21)
      PU L#43 (P#45)
    L2 L#22 (256KB) + L1d L#22 (32KB) + L1i L#22 (32KB) + Core L#22
      PU L#44 (P#22)
      PU L#45 (P#46)
    L2 L#23 (256KB) + L1d L#23 (32KB) + L1i L#23 (32KB) + Core L#23
      PU L#46 (P#23)
      PU L#47 (P#47)
```

```
(Ubuntu) # apt-get install numactl
(RedHat) # yum install numactl

# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 24 25 26 27 28 29 30 31 32 33 34 35
node 0 size: 128811 MB
node 0 free: 89428 MB
node 1 cpus: 12 13 14 15 16 17 18 19 20 21 22 23 36 37 38 39 40 41 42 43 44 45 46
47
node 1 size: 129019 MB
node 1 free: 92592 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10

# lscpu | grep NUMA
NUMA node(s):          2
NUMA node0 CPU(s):     0-11,24-35
NUMA node1 CPU(s):     12-23,36-47
```

To check NIC assignment to NUMA node. We recommend assigning the Cores (real and sibling/hyperthreaded) to the same NUMA as the NIC(s).

```
# cat /sys/class/net/eth0/device/numa_node
0
```

Use lspci command to find PCI addresses of NICs that will be bound to DPDK process

```
# lspci | grep Ethernet
02:00.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5719 Gigabit
Ethernet PCIe (rev 01)
02:00.1 Ethernet controller: Broadcom Corporation NetXtreme BCM5719 Gigabit
Ethernet PCIe (rev 01)
02:00.2 Ethernet controller: Broadcom Corporation NetXtreme BCM5719 Gigabit
Ethernet PCIe (rev 01)
02:00.3 Ethernet controller: Broadcom Corporation NetXtreme BCM5719 Gigabit
Ethernet PCIe (rev 01)
05:00.0 Ethernet controller: Intel Corporation Ethernet Controller X710 for 10GbE
SFP+ (rev 01)
05:00.1 Ethernet controller: Intel Corporation Ethernet Controller X710 for 10GbE
SFP+ (rev 01)
0b:00.0 Ethernet controller: Intel Corporation Ethernet Controller X710 for 10GbE
SFP+ (rev 01)
0b:00.1 Ethernet controller: Intel Corporation Ethernet Controller X710 for 10GbE
SFP+ (rev 01)
```

Use dpdk_nic_bind.py command to verify proper binding of NICs to DPDK

```
sudo /opt/contrail/bin/dpdk_nic_bind.py -s
```

Enable Jumbo

Host

To enable Jumbo frames on tap and physical interfaces assign Jumbo on vhost0 interface.

```
# ifconfig vhost0 mtu 9000
```

On DPDK node the mtu needs to be set on vhost0 and it will update the pmd.

Guest

Before Contrail 3.2.8 and 4.0.1 on DPDK Compute:

1. If it is a DPDK application then jumbo on the Guest interface(s) works (except i40e NICs)
2. If it is a Kernel application (Linux Socket Application) then jumbo on the guest interface(s) works from Contrail 3.2.8 / 4.0 (mergeable buffers). For more information, this code enhancement is tracked under https://bugs.launchpad.net/juniperopenstack/+bug/1592935.

Configure driver and network settings

Add parameters to configuration file /etc/contrail/contrail-vrouter-agent.conf

```
[DEFAULT]
platform = dpdk
physical_uio_driver = $dpdk_nic_driver
physical_interface_address = $dpdk_dev_pci;
physical_interface_mac = $dpdk_dev_mac;
```

$dpdk_dev_pci is an interface pci address (in case of bond it is 0000:00:00.0)

$dpdk_dev_mac is a mac address assigned to interface (in case of bond the lower slaves mac)

$dpdk_nic_driver as described in the section “UIO / VFIO” choose linux driver (default is igb_uio)

Configure OpenStack

Configure nova scheduler

On every controller, add values AggregateInstanceExtraSpecFilter and NUMATopologyFilter to the scheduler_default_filters parameter in /etc/nova/nova.conf. Restart nova-scheduler.

Configure host aggregates:

```
openstack aggregate create dpdk
openstack aggregate add host dpdk [hypervisor 1]
openstack aggregate add host dpdk [hypervisor 2]
openstack aggregate set --property dpdk=true dpdk
```

Define flavors for VNFs

```
openstack flavor create --ram 8192 --disk 20 --vcpus 4 --public my-flavor
openstack flavor set --property aggregate_instance_extra_specs:dpdk=true
my-flavor
openstack flavor set --property hw:mem_page_size=large my-flavor
openstack flavor set --property hw:cpu_policy=dedicated my-flavor
openstack flavor set --property hw:cpu_thread_policy=require my-flavor
```

Note that mem_page_size can have 1GB or 2048 values depending on hugepage size requirements of the VNF. Multiple flavors could be created in order to provide for both options.

hw:cpu_policy=**dedicated** - means that Nova will allocate physical cores and their sibling HT

hw:cpu_policy=**isolated** - means that Nova will allocate physical cores only

hw:cpu_policy=**prefered** - allocate range for VM and then system range will allocate cores dynamically. **DO NOT use this option with isolated CPUs!**

PS: **Large pages are required** for each instance **(even any non-DPDK instance) running on hosts with DPDK vrouter**. If large pages are not present in the guest, the **interface will appear but will not function**.

This is why, on a compute node running DPDK vrouter, openstack instances must use flavors having *hw:mem_page_size=large* property.

Spawn a VM

```
openstack server create --image dpdk_image --flavor my-flavor --nic
net-id=276c706c-cf78-44ba-ba97-d9b9dbe36707 vm-dpdk-1
```

Performance

Performance tuning essentials

Juniper Networks recommends to tune performance in the following four areas:

- Enable multiqueue on VNF as described in section How to enable multiqueue.
- Proper CPU allocation as described in section CPU core assignment for vRouter service threads (lcores).
- Not disable Irqbalance as described in section Irqbalance.
- When router is working in Flow based mode, optimize flow setup rate as described in section Flow setup rate optimisation or consider usage of Packed based mode.

The following performance optimization actions are details below:

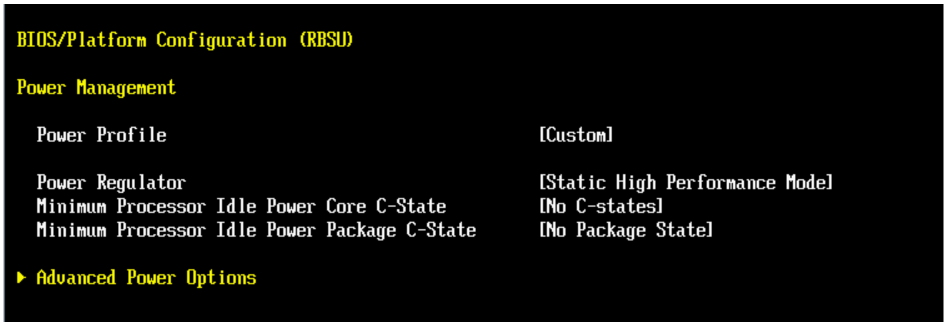
- Enable CPU performance mode
- Manage a high flow environment
- Bond setting
- Change default encapsulation
- DPDK buffer size adjustment (mbuf)
- Jumbo frame support
- Control the number of CPU cores for vRouter

Enable CPU performance mode

CPU frequency scaling enables the operating system to scale the CPU frequency up or down in order to save power. CPU frequencies can be scaled automatically depending on the system load, in response to ACPI events, or manually by userspace programs. To run the CPU at the maximum frequency set scaling_governor to performance.

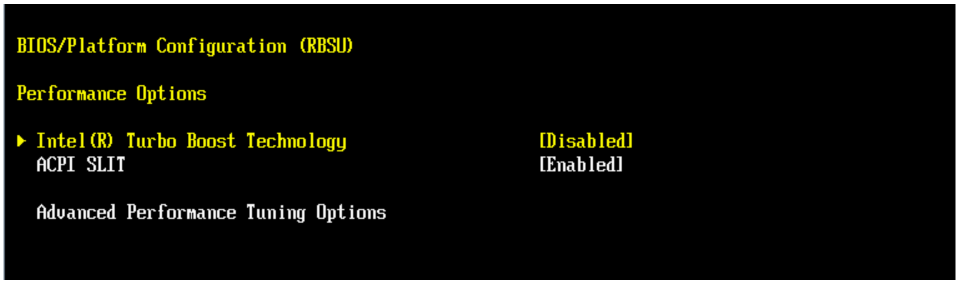
```
for f in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor ; do echo
performance > $f ; cat $f; done
```

Disable all power saving options such as: Power performance tuning, CPU P-State, CPU C3 Report and CPU C6 Report. Select Performance as the CPU Power and Performance policy.



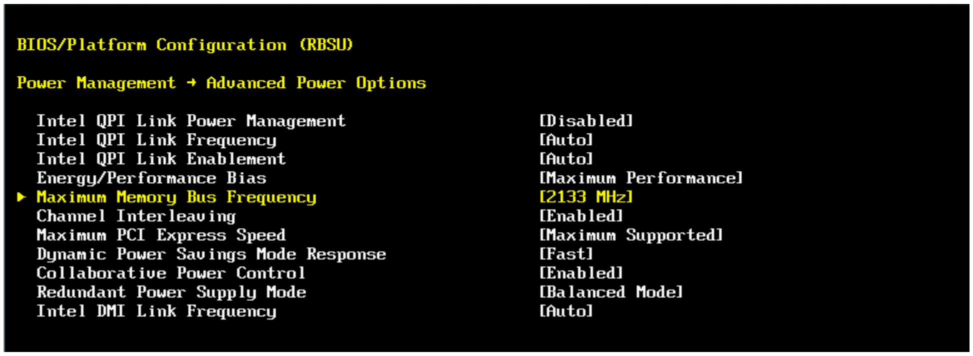
```
BIOS/Platform Configuration (RBSU)

Power Management

  Power Profile                                 [Custom]

  Power Regulator                               [Static High Performance Mode]
  Minimum Processor Idle Power Core C-State     [No C-states]
  Minimum Processor Idle Power Package C-State  [No Package State]

▶ Advanced Power Options
```

Disable Turbo Boost to ensure the performance scaling increases with the number of cores.



```
BIOS/Platform Configuration (RBSU)

Performance Options

▶ Intel(R) Turbo Boost Technology               [Disabled]
  ACPI SLIT                                     [Enabled]

  Advanced Performance Tuning Options
```

Set memory frequency to the highest available number, NOT auto.



```
BIOS/Platform Configuration (RBSU)

Power Management → Advanced Power Options

  Intel QPI Link Power Management               [Disabled]
  Intel QPI Link Frequency                      [Auto]
  Intel QPI Link Enablement                     [Auto]
  Energy/Performance Bias                       [Maximum Performance]
▶ Maximum Memory Bus Frequency                  [2133 MHz]
  Channel Interleaving                          [Enabled]
  Maximum PCI Express Speed                     [Maximum Supported]
  Dynamic Power Savings Mode Response           [Fast]
  Collaborative Power Control                   [Enabled]
  Redundant Power Supply Mode                   [Balanced Mode]
  Intel DMI Link Frequency                      [Auto]
```

Manage a high flow environment

Increase flow table size

The default flow table size is 512K and can be increased by setting the -–vr_flow_entries in /etc/contrail/supervisor-vrouter_files/contrail-vrouter-dpdk.ini (Since Contrail 5.0 release and later, Contrail is now containerized and DPDK parameters are no more defined into contrail-vrouter-dpdk.ini file but are installed by /bin/bash /entrypoint.sh into contrail-vrouter-agent-dpdk container - cf appendix section)

To increase the flow table to 2M flows set the –vr_flow_entries as below:

```
command=taskset 0xff /usr/bin/contrail-vrouter-dpdk --no-daemon --vr_flow_entries=2000000 –vdev
"eth_bond_bond0,mode=4,xmit_policy=l34,socket_id=0,mac=90:e2:ba:b8:70:c4,slave=0000:03:00.0,sla
ve=0000:83:00.0,slave=0000:83:00.1,slave=0000:03:00.1" --socket-mem 1024,1024
```

Then reload kernel module + restart vrouter

Flow setup rate optimisation

For optimal flow setup rate, the agent flow thread count can be increased to 4 (default 2 threads).

```
/etc/contrail/contrail-vrouter-agent.conf
[FLOWS]
thread_count = 4
```

Then run “service supervisor-vrouter restart” for the changes to take effect.

This setting is also available at orchestration level through the following variable in TripleO template:

```
  ContrailSettings:
      VROUTER_GATEWAY: 172.30.254.1
      BGP_ASN: 64512
      BGP_AUTO_MESH: true
      LACP_RATE: 1
      VROUTER_AGENT__FLOWS__thread_count: '4'
```

To set about the total amount of threads assigned to the agent (not only for flows), the number of threads for agent should be higher than for the flow thread number.

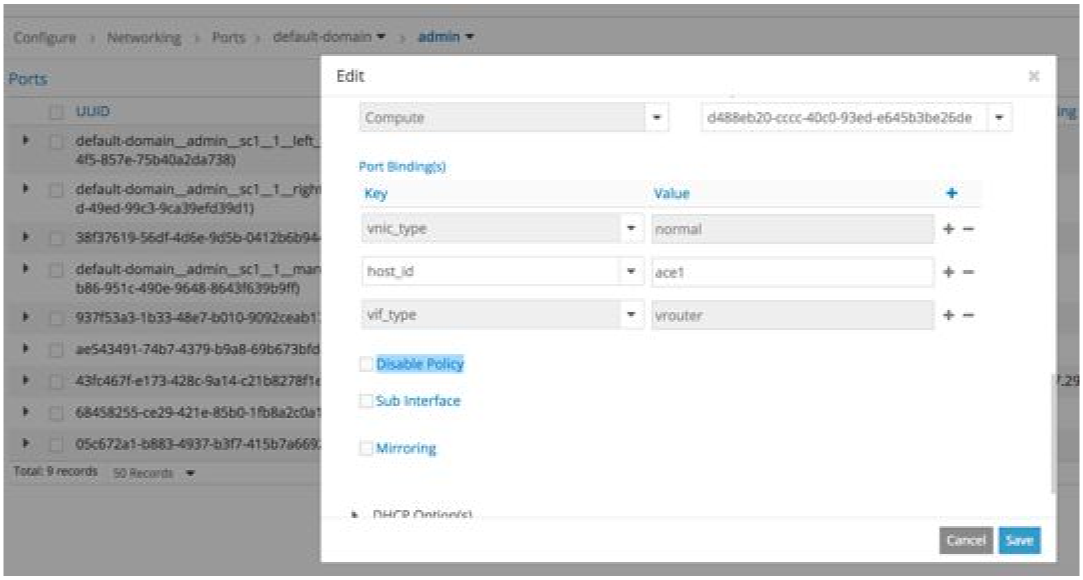
```
/etc/contrail/supervisord_vrouter.conf
environment=TBB_THREAD_COUNT=6
```

Then run “service supervisor-vrouter restart” for the changes to take effect.

Configuration of packet based processing

By default Contrail works in flow-based processing providing a vast feature portfolio. Alternatively, packet based mode can be enabled in Contrail on per interface basis, in order to increase the packet processing rate for traffic that may not need a processing per flow.

Packet-based processing is configurable at port level by selecting the “Disable Policy” checkbox: this will stop the creation of new flows for this interface only.



Configure › Networking › Ports › default-domain › admin

Ports

- □ UUID
- □ default-domain__admin__sc1__1__left_ 4f5-857e-75b40a2da738)
- □ default-domain__admin__sc1__1__righ d-49ed-99c3-9ca39efd39d1)
- □ 38f37619-56df-4d6e-9d5b-0412b6b94
- □ default-domain__admin__sc1__1__man b86-951c-490e-9648-8643f639b9ff)
- □ 937f53a3-1b33-48e7-b010-9092ceab1
- □ ae543491-74b7-4379-b9a8-69b673bfd
- □ 43fc467f-e173-428c-9a14-c21b8278f1e
- □ 68458255-ce29-421e-85b0-1fb8a2c0a1
- □ 05c672a1-b883-4937-b3f7-415b7a669

Total: 9 records 50 Records

Edit

Compute d488eb20-cccc-40c0-93ed-e645b3be26de

Port Binding(s)

Key	Value
vnic_type	normal
host_id	ace1
vif_type	vrouter

□ Disable Policy

□ Sub Interface

□ Mirroring

Despite disabling policy the following features will continue to be available:

- Virtualisation and routing with ECMP
- Metadata service
- Link local
- BGPaaS
- Port Mirroring
- Service Chaining with network policy with rule any any and a single-step chain (no chaining of several SIs).

vrouter physical bond setting

For optimal performance, bond mode must be set to 4, and hash set to layer 3/4 in contrail-vrouter-dpdk.ini on the physical interface (Since Contrail 5.0 release and later, Contrail is now containerized and DPDK parameters are no more defined into contrail-vrouter-dpdk.ini file but are installed by /bin/bash /entrypoint.sh into contrail-vrouter-agent-dpdk container - cf appendix section).

```
/etc/contrail/supervisord_vrouter_files/contrail-vrouter-dpdk.ini


[program:contrail-vrouter-dpdk]
command=/usr/bin/taskset 0xff /usr/bin/contrail-vrouter-dpdk --no-daemon
--vr_flow_entries=2000000 --vdev
"eth_bond_bond0,mode=4,xmit_policy=l34,socket_id=0,mac=90:e2:ba:c5:79:90,s
lave=0000:01:00.0,slave=0000:01:00.1,slave=0000:02:00.0,slave=0000:02:00.1"
--socket-mem 1024,1024
priority=410
autostart=true
killasgroup=true
stdout_capture_maxbytes=1MB
redirect_stderr=true
stdout_logfile=/var/log/contrail/contrail-vrouter-dpdk-stdout.log
stderr_logfile=/var/log/contrail/contrail-vrouter-dpdk-stderr.log
exitcodes=0                   ; 'expected' exit codes for process (default
0,2)
```

Then run “service supervisor-vrouter restart” for the changes to take effect.

PS: Bond linux setup “lacp bond mode=4,xmit_policy=l23” is performing load-balancing on **layer2+layer3 outer header**. In other words, outgoing packets are spread onto both interface according underlay destination IP (as source IP is the same for all encapsulated packets sent by a given compute).

Every packet sent by a compute node to a given remote compute node will use the same outgoing Ethernet interface for a given encapsulation protocol (VxLAN, MPLSoUDP or MPLSoGRE).

Using Bond linux setup “lacp bond mode=4,xmit_policy=l34” is performing load-balancing on **layer3+layer4 outer header.** With this setup when a UDP encapsulation protocol is used (MPLSoUDP or VxLAN), outgoing packets to be sent to a given compute node are spread onto both outgoing Ethernet interfaces according to UDP source port that is used to create traffic “entropy” (diversity) between different flows.

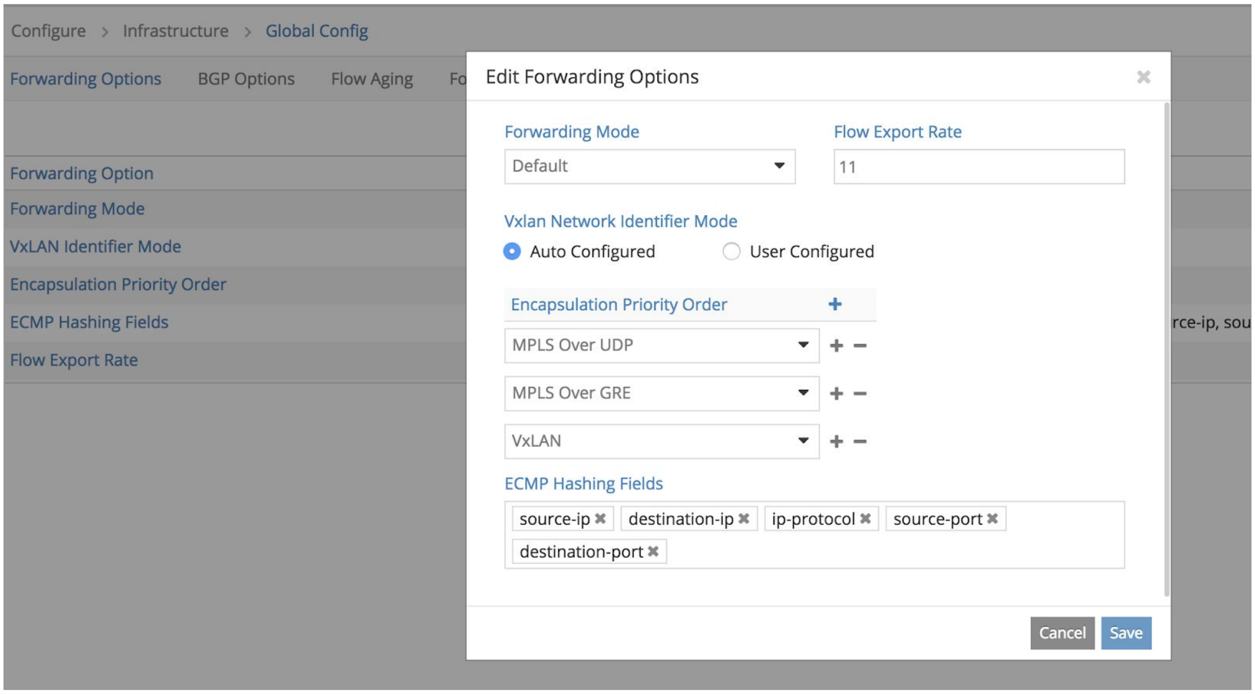
MPLSoUDP encapsulation Configuration

Contrail Configuration

During provisioning of Contrail sometimes provisioning tool not set an encapsulation priority by this for communication to gateways and among vrouters is using default MPLSoGRE. MPLSoGRE is less efficient and not support 5 tuple entropy. To change it encapsulation priority need to be set as follow:

1. MPLSoUDP
2. MPLSoGRE
3. VxLAN

The encapsulation can be set by WebUI.



Configure > Infrastructure > Global Config

Forwarding Options   BGP Options   Flow Aging   Fo

Forwarding Option
Forwarding Mode
VxLAN Identifier Mode
Encapsulation Priority Order
ECMP Hashing Fields   rce-ip, sou
Flow Export Rate

Edit Forwarding Options ✖

Forwarding Mode: Default

Flow Export Rate: 11

Vxlan Network Identifier Mode: ● Auto Configured ○ User Configured

Encapsulation Priority Order +
1. MPLS Over UDP + −
2. MPLS Over GRE + −
3. VxLAN + −

ECMP Hashing Fields: source-ip ✖ destination-ip ✖ ip-protocol ✖ source-port ✖ destination-port ✖

Cancel   Save

SDN Gateway configuration: MPLSoUDP and Load Balancing

MPLSoUDP configuration

In parallel, for optimal load balancing capabilities and resource usage optimisation the SDN gateway must also be configured with the MPLSoUDP encapsulation.

The below configuration applies for JUNOS devices (MX). Note that Junos MPLSoUDP support starts with 16.2 onwards.

Step 1: Configure Tunnels with MPLSoUDP encapsulation

```
[edit routing-options]
autonomous-system 65100;
dynamic-tunnels {
  contrail {
    source-address 10.0.0.100;
    udp; # <===== configuration MPLSoUDP encapsulation (data plane)
    destination-networks {
       10.0.0.0/24;
    }
  }
}
```

Step 2: Configuration of Control Plane signalling (BGP) to advertise the MPLSoUDP encapsulation community (RFC 5512). This overrides the Contrail default GRE forwarding mode.

```
# Define a policy and an encapsulation community (RFC 5512) to advertise that
prefixes must be reached via an MPLSoUDP Tunnelling encapsulation.
[edit policy-options]
community com-encaps-udp members 0x030c:65100:13; #last digit MUST be 13 (Type =
MPLSoUDP)
policy-statement pol-set-udp-encaps { # define a policy
  then {
    community add com-encaps-udp;
  }
}

# Apply this policy toward the MP-BGP peering toward the Contrail Control Nodes.

[edit protocols bgp]
group to-Cluster1 {
  export pol-set-udp-encaps; # apply policy adding UDP encap community
  vpn-apply-export; # if routes exported from a local VRF is defined on PE
  […]
  neighbor 10.0.0.6;
}
```

Virtual Network Load Balancing Configuration

In parallel, some VNF design can also require the activation of Load Balancing (a.ka. multipath) at VRF level.

```
# Control Plane ECMP activation
routing-instances {
  VRF_XXXXX {
    routing-options {
      multipath;
    }
  }
}
# Data Plane ECMP activation
routing-options {
  forwarding-table {
    export load-balance-per-flow;
  }
}
policy-options {
  policy-statement load-balance-per-flow {
    term 1 {
      then {
        load-balance per-packet;
        accept;
      }
    }
  }
}
# Optional GTP TEID hashing activation for mobile use cases
forwarding-options {
       enhanced-hash-key {
       family inet {
             gtp-tunnel-endpoint-identifier;
       }
   }
}
```

DPDK buffer size adjustment (mbuf)

Received packets in Fortville NICs are processed differently to Niantic. Contrail vRouter adjusts to Intel recommendations and increases the number of TX and RX buffer descriptors from 128 to 2048 and the mbuf pool from 16K to 32K bytes.

Starting from the Contrail 3.2.10 release it will be possible to configure the number of TX / RX descriptors in contrail-vrouter-dpdk.ini file (https://bugs.launchpad.net/juniperopenstack/+bug/1743391).

Since Contrail 5.0 release and later, Contrail is now containerized and DPDK parameters are no more defined into contrail-vrouter-dpdk.ini file but are installed by /bin/bash /entrypoint.sh into contrail-vrouter-agent-dpdk container - cf appendix section.

Caveat: The Intel NIC may silently drop packets in case of a shortage of RX descriptors. The DPDK vRouter polls NIC for these statistics.

Intel recommendation to reduce or avoid packet loss

When using Niantic (PF), the packets can be either buffered in the RX descriptors (or, more correctly speaking, in the memory addressed by the RX descriptors), or in the RX packet buffer in the NIC. RX packet buffer size is 512KB when flow director is disabled, hence it can hold > 8000 packets of 64 Bytes (=> *handle an interrupt longer than 500 microseconds - https://etherealmind.com/wp-content/uploads/2017/01/X520_to_XL710_Tuning_The_Buffers.pdf)*.

When using Fortville, packets are by default dropped when RX descriptors are not available. Hence, by default, the only buffers available to store packets when a core is interrupted are the buffers pointed to by the RX descriptors.

A similar behavior (packets dropped when no descriptors available) is obtained on Niantic when using Virtual functions: to avoid “head of line” blocking, packets are dropped if no descriptor is available and virtual functions are used.

So, to avoid packet losses due to CPU core being interrupted when using Fortville (*or when using Niantic and **SR-IOV***), the number of RX descriptors should be configured high enough, for instance to 2048.

Setup number of rx descriptors through rte_eth_rx_queue_setup (see http://www.dpdk.org/browse/dpdk/tree/examples/l3fwd/main.c#n990)

```
ret = rte_eth_rx_queue_setup(portid, queueid, nb_rxd, socketid, NULL,
pktmbuf_pool[socketid]);
```

The number of TX descriptors should also be considered and depends on the application. Imagine the CPU core is interrupted and up to 2048 packets are now buffered through the RX descriptors. Once the interrupt is dealt with, up to 2048 packets can be received very fast by the application: as the packets are already in memory, they might arrive much faster than 10Gbps. Hence, when the application tries to forward those packets, the application might try to send those packets faster than 10Gbps. Some of those packets must be buffered before being sent to the 10 Gb/s link. Either the application takes care of this by buffering those packets, or those packets must be buffered in the TX descriptors. In that case, the number of TX descriptors should also be increased (e.g. up to 2048).

Increasing the number of RX and/or TX descriptors implies that you might have to increase the number of pre-allocated mbufs. This can have a small impact on performance (throughput), as more memory is being used.

Setup number of mbufs through rte_pktmbuf_pool_create: (see
http://www.dpdk.org/browse/dpdk/tree/examples/l3fwd/main.c#n724)

```
rte_pktmbuf_pool_create(s, nb_mbuf, MEMPOOL_CACHE_SIZE, 0,

RTE_MBUF_DEFAULT_BUF_SIZE, socketid);
```

Problem with reassigning kernel driver back to interface when DPDK vRouter is stopped

On Compute node In file /opt/contrail/bin/vrouter-functions.sh (line 532) in s/\x1([0-9:.]+)[\x1]+/ \1/g there are missing letters from a to f. This should be corrected and that could impact on not binding interface back to interface kernel driver from UIO kernel driver when vRouter is stopped.

```
## Look for slave PCI addresses of vRouter --vdev argument
DPDK_BOND_PCIS=`sed -nr \
    -e '/^ *command *=/ {
        s/slave=/\x1/g
        s/[^\x1]+//
        s/\x1([0-9:\.]+)[^\x1]+/ \1/g
        p
    }' \
    ${VROUTER_DPDK_INI}`
```

The correct sed should have letters s/\x1([0-9**a-f**:\.]+)[^\x1]+/ \1/g.

This issue will be solved in release 5.0.1 (https://bugs.launchpad.net/juniperopenstack/+bug/1774403).

Jumbo Frame support

There are limitations with jumbo frame support for Contrail Versions below 4.0.1 and 3.2.8:

DPDK Compute:

- if it is a DPDK application then jumbo on the Guest interface(s) WORKS.
- if it is a Kernel application (Linux Socket Application) then jumbo on the guest interface(s) DO NOT WORK.

Kernel Compute:

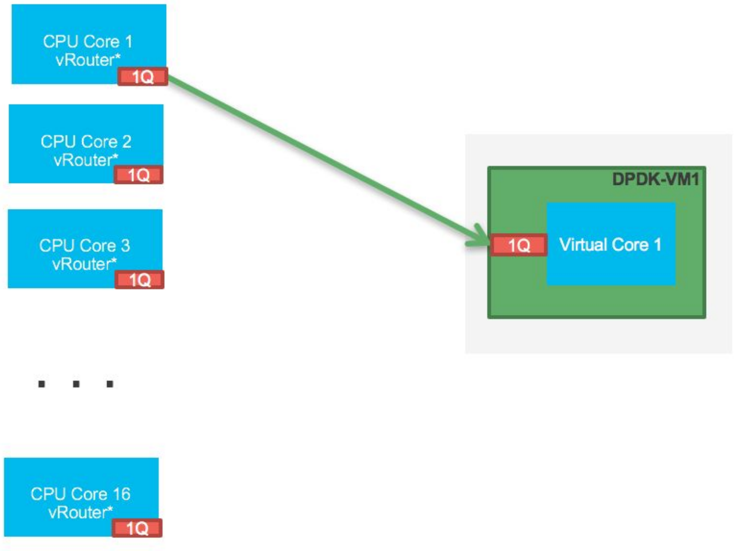
- if it is a Kernel application (Linux Socket Application) then jumbo on the Guest interface(s) works.

This issue is tracked under https://bugs.launchpad.net/juniperopenstack/+bug/1731174. In summary, a DPDK-enabled vRouter does not support jumbo MTU at VNIC for non-DPDK VM (for example, a sockets based application that uses the guest kernel network stack). The VM only posts 1500 byte buffers in this scenario if segmentation offloads are disabled. We do not support these offloads yet inside vrouter-dpdk.

Starting from Contrail 4.0.1 and 3.2.8, jumbo frames are supported in all scenarios.

Control the number of CPU cores vRouter

In non-multiqueue VNF with one DPDK interface, increasing the number of CPU cores for vRouter can actually decrease performance because of One to Many CPU cores queues are sending packets one by one what takes time and can produce microbursts.



On the diagram cores one by one are sending traffic to VM and when last finish data processing potentially first will have no time (buffers) to send VM (virtio queue).

During this case microbursts affecting VNF will be observed on vRouter as TX errors.

Appendixes

vrouter fine tuning parameters (kernel and DPDK mode)

Generic vrouter dimensioning parameters

Since Contrail 5.0 release and later, Contrail is containerized. Kernel vrouter module is still using /etc/modprobe.d/vrouter.conf file to get specific dimensioning values. These parameters are**:**

- **vr_flow_entries**: maximum flow entries (default is 512K)
- **vr_oflow_entries**: maximum overflow entries (default is 8K)
- **vr_bridge_entries**: maximum bridge entries (default is 256K)
- **vr_bridge_oentries**: maximum bridge overflow entries
- **vr_mpls_labels**: maximum MPLS labels used in the node (default is 5K)
- **vr_nexthops**: maximum next hops in the node (defaut is 512K since contrail 19.11 - 65K before)
- **vr_vrfs**: maximum VRFs supported in the node
- **vr_interfaces**: maximum interfaces that can be created (default is 4352); this can be modified from release 3.1.1.0 & 3.2
- **vrouter_dbg**: 1 to dump packets, 0 to disable (disabled by default)

On a containerized **Kernel mode vrouter**, in order to be able to define a specific value for these parameters we have to **create** or edit /etc/modprobe.d/vrouter.conf configuration file onto the compute host using the following command:

```
$ vi /etc/modprobe.d/vrouter.conf
```

Once done the vrouter interface has to be restarted.

```
$ ifdown vhost0
$ ifup vhost0
```

And the vrouter agent container has also to be started:

```
$  docker start contrail_vrouter_agent
```

vRouter parameters can be checked with :

```
$ docker exec contrail_vrouter_agent vrouter --info | more
```

PS: It’s also possible to define vrouter option values into ***VROUTER_MODULE_OPTIONS*** environment variable into */etc/sysconfig/network-scripts/ifcfg-vhost0* file.
For instance, we can add the following line to this file to decrease the Next Hop limit:

```
VROUTER_MODULE_OPTIONS="vr_nexthops=32768"
```

Then we are restarting vhost0 interface and vrouter agent container in order to get it applied.

On a containerized **DPDK mode vrouter**, in order to be able to define a specific value for these parameters we have to use ***DPDK_COMMAND_ADDITIONAL_ARGS*** variable into /etc/sysconfig/network-scripts/ifcfg-vhost0.
/!\: the syntax to be used is slightly different as for the Kernel mode vrouter. All parameters have to be prefixed with “--”. eg: DPDK_COMMAND_ADDITIONAL_ARGS=”--vr_nexthops=32768”

Once done the vrouter interface has to be restarted.

```
$ ifdown vhost0
$ ifup vhost0
```

PS: No need to start the vrouter agent container with the DPDK vrouter:

vRouter parameters can be checked with :

```
$ docker exec contrail_vrouter_agent vrouter --info | more
```

<u>Example</u>:

We will decrease the flow table size from 512 kB to 400 kB onto a **Kernel Mode** vrouter. Before the change we can display the flow table size using vrouter --info command:

```
$ sudo docker exec vrouter_vrouter-agent_1 vrouter --info | grep -i flow\ table
    Flow Table limit              524288
    Flow Table overflow limit        105472
```

vr_flow_entries parameter is set to 400000 into the vrouter.conf configuration file onto the host compute:

```
$ vi /etc/modprobe.d/vrouter.conf
options vrouter vr_flow_entries=400000
```

Then we are restarting vhost0 interface :

```
$ ifdown vhost0
$ ifup vhost0
$  docker start contrail_vrouter_agent
```

After the change we can notice the new value using vrouter --info command :

```
$ sudo docker exec vrouter_vrouter-agent_1 vrouter --info | grep -i flow\ table
    Flow Table limit              400000
    Flow Table overflow limit        80896
```

If you are facing an issue with the router agent container which is not succeeding to go up, it's probably due to a lack of memory.

You can try to free unused blocks using the following procedure :

```
$ sudo -i
# ifdown vhost0
# rmmod vrouter
# free && sync && echo 3 > /proc/sys/vm/drop_caches && free
# modprobe vrouter
# ifup vhost0
```

Cf: https://github.com/Juniper/contrail-controller/wiki/Vrouter-Module-Parameters

WARNING: Increase vrouter default value of dimensioning parameters (especially flow table)

Default parameters of vrouter have to be changed carefully. Some side effects could happen when changing this parameters, especially on system memory consumption.

For instance, in the previous section it has been described how to change vrouter setup and how to increase default size value for many table used by the vrouter. In some situations it could be needed to increase the number of hugepages allocated to the system.

For instance, we can sustain around 12 000 flows per 2M hugepage. When not enough huge pages are available, vhost0 interface can't be created at the vrouter startup:

```
[root@overcloud-contraildpdk-0 ~]# ifup vhost0
b3a25600ece1637e82e017b01357826e2c0abd29e64e8464c67fc5ff61512732
INFO: wait DPDK agent to run... 1
INFO: wait DPDK agent to run... 2
INFO: wait DPDK agent to run... 3
INFO: wait DPDK agent to run... 4
...
```

We can see the following error messages into vrouter dpdk log file :

```
2019-06-19 14:59:02,619 VROUTER: Error mapping file /dev/hugepages2M/flow: Cannot allocate memory (12)
2019-06-19 14:59:02,619 VROUTER: Error initializing flow table: Cannot allocate memory (12)
```

Number of allocated huge pages can be checked with following command:

```
# cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
```

Number of free huge pages can be checked with the following one :

```
# cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/free_hugepages
```

When the number of available hugepages fall to 0 (or is below to the needed value by the vrouter), we have to increase the number of hugepages allocated at the system startup.

In order to do so, we have to change GRUB config.

Here we are setting 64 x 2M hugepages on the system:

```
# vi /etc/default/grub
....
TRIPLEO_HEAT_TEMPLATE_KERNEL_ARGS=" default_hugepagesz=1GB hugepagesz=1G hugepages=10 hugepagesz=2M hugepages=64"
GRUB_CMDLINE_LINUX="${GRUB_CMDLINE_LINUX:+$GRUB_CMDLINE_LINUX }${TRIPLEO_HEAT_TEMPLATE_KERNEL_ARGS}"
```

Once this file has been changed, we have to rebuild the grub config using this command (RedHat):

```
grub2-mkconfig -o /boot/grub2/grub.cfg
```

Once done, we are rebooting the node. After reboot, the number of hugepages can be checked with following commands:

```
# cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-3.10.0-957.10.1.el7.x86_64 root=UUID=334f450f-1946-4577-a4eb-822bd33b8db2 ro console=tty0 console=ttyS0,115200n8 crashkernel=auto rhgb quiet default_hugepagesz=1GB hugepagesz=1G hugepages=10 hugepagesz=2M hugepages=64
```

```
# cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
64
```

If we want to be able to sustain 2 000 000 flows, we need to have around 164 pages (2 000 000 / 12 000). With the procedure seen earlier we are changing the number of hugepages allocated to the system:

```
[root@overcloud-contraildpdk-0 ~]# cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
164
```

Then, *vr_flow_entries* parameter is set to 2 000 000 into the vrouter.conf configuration file inside the vrouter agent container :

```
$ sudo docker exec -it vrouter_vrouter-agent_1 vi /etc/modprobe.d/vrouter.conf
options vrouter vr_flow_entries=2000000
```

Then we are restarting vhost0 interface :

```
$ ifdown vhost0
$ ifup vhost0
```

This last command should be successful:

```
[root@overcloud-contraildpdk-0 ~]# ifup vhost0
72acf9590f7bbb76cf737c535e20665ace495c9cc5783d2014f8778363fef221
INFO: wait DPDK agent to run... 1
INFO: wait DPDK agent to run... 2
INFO: wait vhost0 to be initilaized... 0/60
INFO: wait vhost0 to be initilaized... 1/60
INFO: vhost0 is ready.
```

Once done we can see that the number of available hugepages has fallen near to 0:

```
[root@overcloud-contraildpdk-0 ~]# cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/free_hugepages
7
```

And we can see the flow table size has been set to 2 000 000 :

```
[root@overcloud-contraildpdk-0 ~]# sudo docker exec -it contrail-vrouter-agent-dpdk vrouter --info | grep -i flow\ table
Flow Table limit              2000000
Flow Table overflow limit         400384
```

32 bits vrouter nexthop limit parameter (contrail release 19.11 and later)

In release 19.10 and earlier next hop id in contrail virtual router was 16 bits. Default value for next hop limit was set to 65536. In high-scaled environment next hop limit would often exceed leading to traffic drops. After reaching next hop limit contrail vrouter agent still sent new next hop adds to vrouter, on receiving new config. Failure in creating next hop did not raise any alarm. There was no accounting for usage of vrouter parameters like next hop, mpls label, vrf (cf https://github.com/Juniper/contrail-controller/wiki/Vrouter-Module-Parameters).

Since 19.11 release next hop id in contrail virtual router is increased to 32 bits. New default value of next hop limit is set to 512k (512*1024). Now, contrail vrouter can support up to 1M next hops. New specific alarms are raised once next hop limit exceeds and or once mpls label limit exceeds.

A watermark can be configured in agent configuration file (percentage of vr limits - values: [50-95]).
High Watermark is defined in /etc/contrail/*contrail-vrouter-agent.conf*. Low watermark in agent is high watermark – 5.

Example:

```
[DEFAULT]
vr_object_high_watermark = 80
```

Nexthop and mpls label alarm is raised when count for these objects reaches one of these limits:

- watermark*vr_nexthops
- watermark*vr_mpls_labels

Since 19.11 release, two new alarms have been created :

- **system-defined-vrouter-limit-exceeded alarm** (severity major)
  This alarm is raised when nexthop or mpls labels count crosses high watermark.
  This alarm is cleared when nexthop and mpls labels count becomes lower than the low watermark.
- **system-defined-vrouter-table-limit-exceeded** (severity critical)
  This alarm is raised when nexthop or mpls labels count reaches nexthop or mpls labels count configured in vrouter.
  This alarm is cleared once nexthop or mpls labels count goes below 95% of nexthop and mpls labels count configured in vrouter.

These alarms are raised when either nexthop or mpls labels or both have exceeded the limits. The type of resource/table is available in the alarm

vrouter networking parameters (MTU)

Some other parameters (like MTU) are directly set into /etc/sysconfig/network-scripts/ifcfg-vhost0.

```
$ sudo vi /etc/sysconfig/network-scripts/ifcfg-vhost0
MTU=1400
```

Then vhost0 interface has to be restarted :

```
$ ifdown vhost0
$ ifup vhost0
```

PS: */etc/sysconfig/network-scripts/network-functions-vrouter-kernel-env* can also be used.

vrouter DPDK fine tuning parameters

DPDK vrouter specific parameters

Main parameters that should be customized onto DPDK vrouters in order to get better performances are :

- mempool size
- Physical NIC (PMD) TX and RX descriptors size
- socket memory size

vrouter default values are .

- 16384 Bytes for membuf size
- 128 for TX and RX descriptors size

Since 20.03 release, some new parameters have been added in order to be able to configure :

- vrouter forwarding lcores TX and RX descriptors size
- yield deactivation on forwarding cores

vrouters DPDK fine tuning parameters :

--dpdk_ctrl_thread_mask : *(20.03 and later version)* CPUs to be used for vrouter control threads (CPU list or hexadecimal bitmask).

--service_core_mask : *(20.03 and later version)* CPUs to be used for vrouter service threads (CPU list or hexadecimal bitmask).

--yield_option : *(20.03 and later version)* is used to enable or disable yield on forwarding cores (0 or 1 - enabled by default). In the case below, yield is disabled onto forwarding cores.

```
--yield_option 0
```

--vr_dpdk_tx_ring_sz : *(20.03 and later version)* is used to define forwarding lcores TX Ring descriptor size (1024 by default). In the case below, TX Ring descriptor size has been set to 2048.

```
--vr_dpdk_tx_ring_sz 2048
```

--vr_dpdk_rx_ring_sz : *(20.03 and later version)* is used to define forwarding lcores RX Ring descriptor size (1024 by default). In the case below, RX Ring descriptor size has been set to 2048.

```
--vr_dpdk_rx_ring_sz 2048
```

--socket-mem : is used to define the amount of memory pre-allocated for contrail vrouter. In the case below, 1GB of huge-page memory is pre-allocated on NUMA node 0 and NUMA node 1.

```
--socket-mem 1024,1024
```

--vr_mempool_sz : is used to define mempool memory size. In the case below 128 MB mempool memory size is defined.

```
--vr_mempool_sz 131072
```

--dpdk_txd_sz : is used to define Physical NIC TX Ring descriptor size. In the case below 2048 bytes RX ring descriptor size is defined.

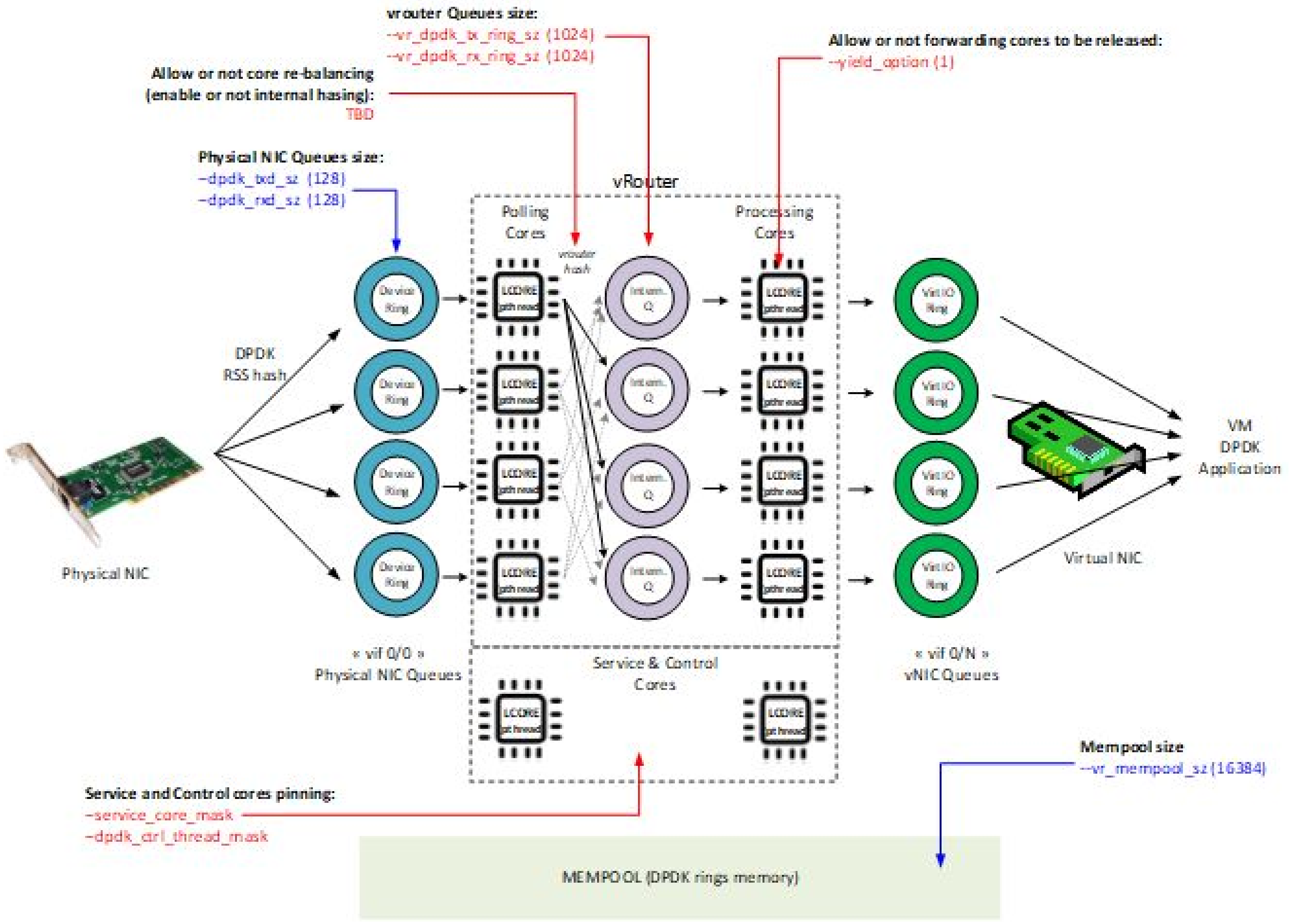
```
--dpdk_txd_sz 2048
```

--dpdk_rxd_sz : is used to define Physical NIC RX Ring descriptor size. In the case below 2048 bytes RX ring descriptor size is defined.

```
--dpdk_rxd_sz 2048
```

These values have to be adjusted depending on :

- the inter NIC model used
- the number of NIC members of vhost0 bond
- the number of logical cores allocated to the vrouter



DPDK fine tuning rules (mainly for Intel NIC)

MEMPOOL SIZE

Following formula has to be used to define the mempool size (set of mbufs) to be configured on vrouter :

mempool size = MAX(16384, *2 * (number_of_RX_descriptors + number_of_TX_descriptors) * number_of_vrouter_cores * number_of_ports_in_dpdk_bond*)

Provided mempool (that are containing membufs) size value is suitable to use in most cases. This value has to be confirmed by tests or monitoring of live traffic. Indeed, memory needs are depending on a lot of factors like : packets size, traffic load, bursts, number of virtual NIC, MTU, ...
In some situations it could be needed to increase mempool size above this calculated size. We can detect such a need onto vif interface counters. When mempool size is too low, we can see “no mbuf” high counters values :

```
$ vif --list | grep -B4 “no mbuf”
vif0/0      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
Type:Physical HWaddr:b8:85:84:44:50:d1 IPaddr:0.0.0.0
Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpEr QOS:-1 Ref:152
RX device packets:24559028  bytes:4969135442 errors:0 no mbufs:231501288288

vif0/1      PMD: vhost0
Type:Host HWaddr:b8:85:84:44:50:d1 IPaddr:192.168.63.40
Vrf:0 Mcast Vrf:65535 Flags:L3DEr QOS:-1 Ref:18
RX device packets:596147  bytes:456776973 errors:0 no mbufs:170086
```

Increasing the mempool size gives lower performance and higher caches misses. So it is preferable not to oversize this value and to adjust it to the real needs.

TX/RX DESCRIPTOR SIZE

For Intel Niantic NIC family (without SR-IOV) used default value for TX and RX descriptor size has to be kept (128). For Intel Fortville NIC family (and Intel Niantic NIC family when SR-IOV is used), TX and RX descriptor size has to be increased to 2048.

	SR-IOV used	RX/TX descriptor size
Niantic	No	128
	Yes	2048
Fortville	No	2048
	Yes	2048

SOCKET MEMORY SIZE

Following formula has to be used to define the socket memory size to be configured on vrouter :

if mempool size <= 131072	then use	**socketmem = 1024**
if mempool size > 131072	then use	**socketmem = 2048**

CPU CORE_MASK

For DPDK CPU allocation (CPU affinity mask definition), it is preferable to follow rules hereafter :

- CPU 0 should always be left for the host process
- CPUs should be associated with NUMA node of the DPDK interface
- CPU siblings (in case of Hyper-threading) should be added together
- CPUs assigned to PMD should be excluded from nova scheduling using NovaVcpuPinset

PS. Following command gives the siblings of a CPU:

```
cat /sys/devices/system/cpu/cpu<cpu no>/topology/thread_siblings_list
```

Example of vrouter dpdk setup with Intel NIC cards (SR-IOV not used) :

	#LCORES	#BOND MEMBERS	TX/RX DESC SIZE	MEMPOOL SIZE	SOCKET MEM
Default Setup	4	N/A	128	N/A	N/A
Niantic Family (82599/X520) 10G interface	4	2	128	16384	1024
	8	2	128	16384	1024
Fortville Family (XL 710) 40G interfaces	4	2	2048	65536	1024
	8	2	2048	131072	1024

Hence, into */etc/contrail/supervisord_vrouter_files/contrail-vrouter-dpdk.ini* we have to put above parameters

4 or 8 logical cores on 2 Niantic bond members - 2 NUMA nodes

```
command=/bin/taskset <CPU 4 or 8 cores affinity> /usr/bin/contrail-vrouter-dpdk
--socket-mem 1024,1024 …
```

4 logical cores on 2 Fortville bond members - 2 NUMA nodes

```
command=/bin/taskset <CPU 4 cores affinity> /usr/bin/contrail-vrouter-dpdk
--dpdk_rxd_sz 2048 --dpdk_txd_sz 2048 --vr_mempool_sz 65536
--socket-mem 1024,1024 …
```

8 logical cores on 2 Fortville bond members - 2 NUMA nodes

```
command=/bin/taskset <CPU 8 cores affinity> /usr/bin/contrail-vrouter-dpdk
--dpdk_rxd_sz 2048 --dpdk_txd_sz 2048 --vr_mempool_sz 131072
--socket-mem 1024,1024 …
```

DPDK vouter parameters provisioning

DPDK fine tuning setup has first to be defined before Contrail rollout into the used OpenStack Contrail deployment tools.

For instance, if TripleO is used, specific parameters have to be defined into *ContrailDpdkOptions* variable. This variable is used to fulfill *DPDK_COMMAND_ADDITIONAL_ARGS* environment variable for DPDK vrouter container instance. If Ansible is used, *DPDK_COMMAND_ADDITIONAL_ARGS* parameter can be used.

For instance with TripleO :
An environment file (*<tripleO root dir>*/environments/contrail/contrail-service.yaml) has to be used to declare *ContrailDpdkOptions* value. With TripleO there are two possibilities:

First : declare this parameter at global level (same value for any DPDK roles):

```
parameter_defaults:
  ContrailDpdkOptions: “--yield_option 0 …”
```

Second : declare this parameter at role level (specific value for each DPDK roles):

```
# role specific variables:
parameter_defaults:
  <Role Name>Parameters:
    ContrailSettings:
         # for service threads
         SERVICE_CORE_MASK: ”0x400000000400”
         # for ctrl threads
         DPDK_CTRL_THREAD_MASK: ”0x2B00000002B”
    ContrailDpdkOptions: “--yield_option 0 …”
```

For instance with Ansible :

```
instances:
  bms1:
    provider: bms
    ip: 192.204.216.44
    roles:
      vrouter:
        AGENT_MODE: dpdk
        SERVICE_CORE_MASK: ”0x400000000400”
        DPDK_CTRL_THREAD_MASK: ”0x2B00000002B”
        CPU_CORE_MASK: ”0x154000000154”
        DPDK_COMMAND_ADDITIONAL_ARGS: “--yield_option 0 …”
```

When contrail 5.0 or later release is used, dpecified valuers for these parameters will be put into */etc/sysconfig/network-scripts/ifcfg-vhost0* config file on each DPDK compute node..

DPDK vouter parameters configuration onto the compute node (5.0 and later versions)

Since Contrail 5.0 release and later, Contrail is containerized and DPDK parameters are no more defined into contrail-vrouter-dpdk.ini file but are installed by */bin/bash /entrypoint.sh* into contrail-vrouter-agent-dpdk container.

At the startup, DPDK vrouter container which will run /bin/bash /entrypoint.sh, which is applying specific DPDK configuration defined into *DPDK_COMMAND_ADDITIONAL_ARGS* environment variable.

When a specific DPDK setup update is required after the initial rollout, it could be defined into /etc/sysconfig/network-scripts/network-functions-vrouter-dpdk-env file. In this file we can override *DPDK_COMMAND_ADDITIONAL_ARGS* value defined into the deployment script.

Example :

We are adding new specific TX and RX ring size (256 bytes instead of 128 bytes default value) and using a specific CPU mask into *network-functions-vrouter-dpdk-env file* :

```
 $ vi /etc/sysconfig/network-scripts/network-functions-vrouter-dpdk-env
#!/bin/bash
DPDK_COMMAND_ADDITIONAL_ARGS="--dpdk_rxd_sz 256 --dpdk_txd_sz 256"
CPU_LIST=0x154
```

PS: Specific DPDK fine tuning could also be defined into */etc/sysconfig/network-scripts/ifcfg-vhost0* file instead of */etc/sysconfig/network-scripts/network-functions-vrouter-dpdk-env* file.

```
$ vi /etc/sysconfig/network-scripts/ifcfg-vhost0
...
DPDK_COMMAND_ADDITIONAL_ARGS="--dpdk_rxd_sz 256 --dpdk_txd_sz 256"
CPU_LIST=0x154
MTU=9000
```

In order to get these new values taken into consideration we have to restart vhost0 interface :

```
$ sudo ifdown vhost0
$ sudo ifup vhost0
```

After vhost0 interface restart the new values have been applied :

```
$ sudo ps -ef | grep vrouter-dpdk
root       97244   97226  0 16:30 ?            00:00:00 /bin/bash /entrypoint.sh /usr/bin/contrail-vrouter-dpdk
root       97476   97244 99 16:30 ?            00:01:13 /usr/bin/contrail-vrouter-dpdk --no-daemon --dpdk_rxd_sz 256 --dpdk_txd_sz
256 --socket-mem 1024 1024 --vlan_tci 20 --vlan_fwd_intf_name bond0 --vdev
eth_bond_bond0,mode=4,xmit_policy=l23,socket_id=0,mac=3c:fd:fe:bc:f7:e8,lacp_rate=0,slave=0000:03:00.0,slave=0000:03:00.1
heat-ad+   97795   92566  0 16:30 pts/9        00:00:00 grep --color=auto vrouter-dpdk
```

We can also see into contrail dpdk vrouter logs, the new values in use :

```
$ vi /var/log/containers/contrail/contrail-vrouter-dpdk.log
...
2019-04-19 16:30:41,411 VROUTER: vRouter version: {"build-info": [{"build-time": "2019-04-11 23:47:32.407280", "build-hostname":
"rhel-7-builder-juniper-contrail-ci-c-0000225573.novalocal", "build-user": "zuul", "build-version": "5.1.0"}]}
2019-04-19 16:30:41,411 VROUTER: DPDK version: DPDK 18.05.1
2019-04-19 16:30:41,427 VROUTER: Log file : /var/log/contrail/contrail-vrouter-dpdk.log
2019-04-19 16:30:41,427 VROUTER: Using VLAN TCI: 20
2019-04-19 16:30:41,427 VROUTER: Bridge Table limit:          262144
2019-04-19 16:30:41,427 VROUTER: Bridge Table overflow limit: 53248
2019-04-19 16:30:41,427 VROUTER: Flow Table limit:            524288
2019-04-19 16:30:41,427 VROUTER: Flow Table overflow limit:   105472
2019-04-19 16:30:41,427 VROUTER: MPLS labels limit:           5120
2019-04-19 16:30:41,427 VROUTER: Nexthops limit:             65536
2019-04-19 16:30:41,427 VROUTER: VRF tables limit:            4096
2019-04-19 16:30:41,427 VROUTER: Packet pool size:            16384
2019-04-19 16:30:41,427 VROUTER: PMD Tx Descriptor size:              256
2019-04-19 16:30:41,427 VROUTER: PMD Rx Descriptor size:              256
2019-04-19 16:30:41,427 VROUTER: Maximum packet size:                 9216
2019-04-19 16:30:41,427 VROUTER: Maximum log buffer size:             200
2019-04-19 16:30:41,427 VROUTER: EAL arguments:
2019-04-19 16:30:41,427 VROUTER:          -n  "4"
2019-04-19 16:30:41,427 VROUTER:  --socket-mem  "1024,1024"
2019-04-19 16:30:41,427 VROUTER:          --vdev
"eth_bond_bond0,mode=4,xmit_policy=l23,socket_id=0,mac=3c:fd:fe:bc:f7:e8,lacp_rate=0,slave=0000:03:00.0,slave=0000:03:00.1"
2019-04-19 16:30:41,427 VROUTER:      --lcores  "(0-2)@(0-71),(8-9)@(0-71),10@2,11@4,12@6,13@8"
2019-04-19 16:30:41,432 EAL: Detected 72 lcore(s)
2019-04-19 16:30:41,432 EAL: Detected 2 NUMA nodes
…
```

But, be careful, after this operation contrail_vrouter_agent is stopped, we have to start it :

```
[heat-admin@oln-compute-dpdk-0 ~]$ sudo docker ps --all
CONTAINER ID       IMAGE                                                            COMMAND              CREATED          STATUS
PORTS              NAMES
300c9d8b9987       10.19.2.1:8787/contrail-nightly/contrail-vrouter-agent-dpdk:5.1.0-0.655-rhel-queens       "/entrypoint.sh /u..."   35 seconds ago
             Up 34 seconds                          contrail-vrouter-agent-dpdk
b936827c6999       10.19.2.1:8787/contrail-nightly/contrail-vrouter-agent:5.1.0-0.538-rhel-queens           "/entrypoint.sh /u..."   8 days ago
             Exited (0) 6 minutes ago                 contrail_vrouter_agent
...
```

In order to start contrail_vrouter_agent container we have to issue the following command :

```
$ sudo docker start contrail_vrouter_agent
```

Connectivity Troubleshooting

How to check vrouter connectivity

Verify if VM was started on proper DPDK Compute node.

```
# openstack server list
...get VM UUID...
# nova show <VM_UUID>
```

Verify if tap interface has vhostuser type.

```
# virsh list
 Id    Name                           State
----------------------------------------------------
 5     instance-000032d2              running

# virsh dumpxml instance-000032d2 | grep -A 6 "interface type"
    <interface type='vhostuser'>
      <mac address='02:7f:8e:16:17:ae'/>
      <source type='unix' path='/var/run/vrouter/uvh_vif_tap7f8e1617-ae' mode='client'/>
      <model type='virtio'/>
      <alias name='net0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
    </interface>
```

Use vif --list to match the tap interface name to a vif identifier.

```
# vif --list
vif0/0  OS: pkt0
          Type:Agent HWaddr:00:00:5e:00:01:00 IPaddr:0
          Vrf:65535 Flags:L3 MTU:1514 Ref:2
          RX packets:6591  bytes:648577 errors:0
          TX packets:12150  bytes:1974451 errors:0
vif0/1  OS: vhost0
          Type:Host HWaddr:00:25:90:c3:08:68 IPaddr:0
          Vrf:0 Flags:L3 MTU:1514 Ref:3
          RX packets:3446598  bytes:4478599344 errors:0
          TX packets:851770  bytes:1337017154 errors:0
vif0/2  OS: p1p0p0 (Speed 1000, Duplex 1)
          Type:Physical HWaddr:00:25:90:c3:08:68 IPaddr:0
          Vrf:0 Flags:L3 MTU:1514 Ref:22
          RX packets:1643238  bytes:1391655366 errors:2812
          TX packets:3523278  bytes:6806058059 errors:0
vif0/4 OS: tap7f8e1617-ae
          Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.0.17
          Vrf:1 Flags:PL3L2D MTU:9160 Ref:23
          RX packets:60  bytes:4873 errors:0
          TX packets:21  bytes:2158 errors:0
```

Use vif --get to check if tap interface is connected to correct RI and TX / RX counters are increasing.

```
# vif --get 4
Vrouter Interface Table

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
       Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2
       D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
       Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload,
Mon=Interface is Monitored
       Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New
Flows, Proxy=MAC Requests Proxied Always

vif0/4      PMD: tap7f8e1617-ae
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.0.17
            Vrf:1 Flags:PL3L2D QOS:-1 Ref:23
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:29  bytes:1218 errors:0
            TX packets:29  bytes:1218 errors:0
            TX port   packets:0 errors:29
            Drops:0
```

Use vifdump command to dump traffic on tap interface.

```
# vifdump -i 4
vif0/4      PMD: tap7f8e1617-ae
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on mon4, link-type EN10MB (Ethernet), capture size 262144 bytes
15:01:17.004641 ARP, Request who-has 192.168.0.17 tell 192.168.0.2, length 28
15:01:19.004610 ARP, Request who-has 192.168.0.17 tell 192.168.0.2, length 28
```

Check contrail-status.

```
# contrail-status -d
```

Use dpdk_nic_bind.py command to verify proper binding of NICs to proper Linux driver.

```
# /opt/contrail/bin/dpdk_nic_bind.py -s
```

vRouter vif queues setup

When the router boots up, it displays a message about its logical cores numbering and pNIC and vNIC queues logical core pinning into *contrail-vrouter-dpdk-stdout.log* file.

First vif to be initialized, is physical NIC, vif 0/0 :

```
2019-02-06 10:15:40,516 VROUTER: Adding vif 0 (gen. 1) eth device 2 PCI 0000:00:00.0 MAC
00:00:00:00:00:00 (vif MAC b8:85:84:44:50:83)
2019-02-06 10:15:40,516 VROUTER: Using 64 TX queues, 4 RX queues
...
2019-02-06 10:15:40,831 VROUTER:     lcore 10 TX to HW queue 0
2019-02-06 10:15:40,831 VROUTER:     lcore 11 TX to HW queue 1
2019-02-06 10:15:40,831 VROUTER:     lcore 12 TX to HW queue 2
2019-02-06 10:15:40,831 VROUTER:     lcore 13 TX to HW queue 3
2019-02-06 10:15:40,831 VROUTER:     lcore  8 TX to HW queue 4
2019-02-06 10:15:40,831 VROUTER:     lcore  9 TX to HW queue 5
2019-02-06 10:15:40,831 VROUTER:     lcore 10 RX from HW queue 0
2019-02-06 10:15:40,831 VROUTER:     lcore 11 RX from HW queue 1
2019-02-06 10:15:40,831 VROUTER:     lcore 12 RX from HW queue 2
2019-02-06 10:15:40,831 VROUTER:     lcore 13 RX from HW queue 3
```

On vif 0/0, 4 queues are configured and pinned to one dedicated “packet processing core”. They are queues 0 to 3, pinned to lcore 10 to 13.

Later on, vNIC vif 0/N are also initialized :

```
2019-02-13 14:37:16,250 VROUTER: Adding vif 3 (gen. 4) virtual device tapc02d2743-e3
2019-02-13 14:37:16,250 VROUTER:     lcore 11 TX to HW queue 0
2019-02-13 14:37:16,250 VROUTER:     lcore 12 TX to HW queue 1
2019-02-13 14:37:16,250 VROUTER:     lcore 13 TX to HW queue 2
2019-02-13 14:37:16,250 VROUTER:     lcore  8 TX to HW queue 3
2019-02-13 14:37:16,250 VROUTER:     lcore  9 TX to HW queue 4
2019-02-13 14:37:16,250 VROUTER:     lcore 10 TX to HW queue 5
2019-02-13 14:37:16,250 VROUTER:     lcore 11 RX from HW queue 0
2019-02-13 14:37:16,250 VROUTER:     lcore 12 RX from HW queue 1
2019-02-13 14:37:16,250 VROUTER:     lcore 13 RX from HW queue 2
2019-02-13 14:37:16,250 VROUTER:     lcore 10 RX from HW queue 3
2019-02-13 14:37:16,251 UVHOST: Adding vif 3 virtual device tapc02d2743-e3
2019-02-13 14:37:16,251 UVHOST:     vif (server) 3 socket tapc02d2743-e3 FD is 76
```

On vif 0/0, 4 queues are configured and pinned to one dedicated “packet processing core”. They are also queues 0 to 3, pinned to lcore 10 to 13. We can notice that cpu to queue mapping is not the same on all vif interfaces.

PS: When the vif is created. it’s important to check it has be done in server mode :

```
2019-09-16 12:32:34,069 UVHOST: Adding vif 3 virtual device tape6b3d698-95
2019-09-16 12:32:34,069 VROUTER: Adding vif 4 (gen. 7) virtual device tap32da3a01-1e
2019-09-16 12:32:34,069 UVHOST:     vif (server) 3 socket tape6b3d698-95 FD is 65
```

In our latest release (5.1 and above), a message like :

```
2019-09-16 12:32:34,069 UVHOST:     vif (client) 3 socket tape6b3d698-95 FD is 65
```

Means the vif interface has not been properly created by the vrouter.

vRouter vif queues activation

First the a connected message is confirming the socket is well connected. Then, as soon as QEMU sending VHOST_USER_SET_VRING_ENABLE (message 18) message, ring is enabled from vrouter side:

```
2019-09-12 12:46:40,445 UVHOST:     connected to /var/run/vrouter/uvh_vif_tapfe589e24-ae for uvhost socket FD 190
2019-09-12 12:46:45,859 UVHOST: Client _tapfe589e24-ae: handling message 1
2019-09-12 12:46:45,859 UVHOST: Client _tapfe589e24-ae: handling message 15
2019-09-12 12:46:45,859 UVHOST: Client _tapfe589e24-ae: handling message 16
2019-09-12 12:46:45,859 UVHOST: Client _tapfe589e24-ae: handling message 17
2019-09-12 12:46:45,860 UVHOST: Client _tapfe589e24-ae: no handler defined for message 3
2019-09-12 12:46:45,860 UVHOST: Client _tapfe589e24-ae: handling message 1
2019-09-12 12:46:45,860 UVHOST: Client _tapfe589e24-ae: handling message 13
2019-09-12 12:46:45,860 UVHOST: Client _tapfe589e24-ae: handling message 13
2019-09-12 12:46:52,274 UVHOST: Client _tapfe589e24-ae: handling message 18
```

As soon as this last message is received, the vrouter interface is active. If this last message is not received from QEMU by Contrail vrouter, the interface will be disabled :

```
2019-09-12 12:47:29,107 UVHOST:     connected to /var/run/vrouter/uvh_vif_tape75222a7-31 for uvhost socket FD 212
2019-09-12 12:47:35,733 UVHOST: Client _tape75222a7-31: handling message 1
2019-09-12 12:47:35,733 UVHOST: Client _tape75222a7-31: handling message 15
2019-09-12 12:47:35,733 UVHOST: Client _tape75222a7-31: handling message 16
2019-09-12 12:47:35,733 UVHOST: Client _tape75222a7-31: handling message 17
2019-09-12 12:47:35,733 UVHOST: Client _tape75222a7-31: no handler defined for message 3
2019-09-12 12:47:35,733 UVHOST: Client _tape75222a7-31: handling message 1
2019-09-12 12:47:35,734 UVHOST: Client _tape75222a7-31: handling message 13
2019-09-12 12:47:35,734 UVHOST: Client _tape75222a7-31: handling message 13
2019-09-12 12:53:04,854 UVHOST: Client _tape75222a7-31: shutdown at message receiving
2019-09-12 12:53:04,854 UVHOST: Client _tape75222a7-31: unmapping 0 memory regions:
2019-09-12 12:53:05,980 UVHOST: Client _tape75222a7-31: unmapping 0 memory regions
```

Here, more than 5 minutes after being connected to the socket, the VHOST_USER_SET_VRING_ENABLE has still not been received by the vrouter. So, the interface is being shut.

Here are the User space VHOST message bound for each ID :

Message	**ID**
VHOST_USER_NONE	0
VHOST_USER_GET_FEATURES	1
VHOST_USER_SET_FEATURES	2
VHOST_USER_SET_OWNER	3
VHOST_USER_RESET_OWNER	4
VHOST_USER_SET_MEM_TABLE	5
VHOST_USER_SET_LOG_BASE	6
VHOST_USER_SET_LOG_FD	7
VHOST_USER_SET_VRING_NUM	8
VHOST_USER_SET_VRING_ADDR	9
VHOST_USER_SET_VRING_BASE	10
VHOST_USER_GET_VRING_BASE	11
VHOST_USER_SET_VRING_KICK	12
VHOST_USER_SET_VRING_CALL	13
VHOST_USER_SET_VRING_ERR	14
VHOST_USER_GET_PROTOCOL_FEATURES	15
VHOST_USER_SET_PROTOCOL_FEATURES	16
VHOST_USER_GET_QUEUE_NUM	17
VHOST_USER_SET_VRING_ENABLE	18
VHOST_USER_MAX	19

PS: Files into following directories have to be investigated in order to check a vif has been properly enabled :

- /var/run/vrouter/ : uvh_tap file has to be present for each virtual instance virtual port
- /var/lib/vrouter/ports : files describing virtual instance port properties

Packet drop troubleshooting

Interface traffic counters

To verify packet and error statistics for all interfaces.

```
# vif --list --rate
Interface rate statistics
------------------------
Interface name                VIF ID                        RX                            TX
                                                     Errors   Packets              Errors   Packets
Physical: bondB               vif0/0                  0        644                  0        888
Host: vhost0                  vif0/1                  0        0                    0        0
Agent: unix                   vif0/2                  0        0                    0        0
Virtual: tapf7d6196c-81       vif0/3                  0        0                    0        0
Virtual: tapbd746390-d8       vif0/4                  0        0                    0        0
Virtual: tap122e8e7b-33       vif0/5                  0        0                    0        0
Virtual: tap48f25122-9e       vif0/6                  0        20                   0        98
Virtual: tap3d147039-e9       vif0/7                  0        25                   0        20
```

Identify vRouter statistics (per core). Use without --core to get total interface statistics.

```
# vif --get 0 --core 10 --rate

Interface rate statistics
-------------------------

Vrouter Interface Table

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
       Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2
       D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
       Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload,
Mon=Interface is Monitored
       Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows,
Proxy=MAC Requests Proxied Always

vif0/0      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
            Type:Physical HWaddr:3c:fd:fe:9c:c2:c5 IPaddr:0.0.0.0
            Vrf:0 Flags:TcL3L2Vp QOS:-1 Ref:46
            Core 10 RX device packets:0  bytes:0 errors:0
            Core 10 RX port   packets:15 errors:0
            Core 10 RX queue  packets:15 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Core 10 RX packets:0  bytes:0 errors:0
            Core 10 TX packets:0  bytes:189 errors:0
```

```
          Core 10 TX queue  packets:0 errors:0
          Core 10 TX port   packets:0 errors:0
          Core 10 TX device packets:0  bytes:0 errors:0
          Drops:8772364
```

Each physical core is mapped to a virtual core starting from 10 to 10+N. (Don’t confuse these lcores with host CPU cores). In the case of MPLSoGRE one core will receive all the pkts and sprays to other cores. Cycle through the lcores from 10 to 10+(N-1) where N is the no of cores assigned to vrouter.

Counters description

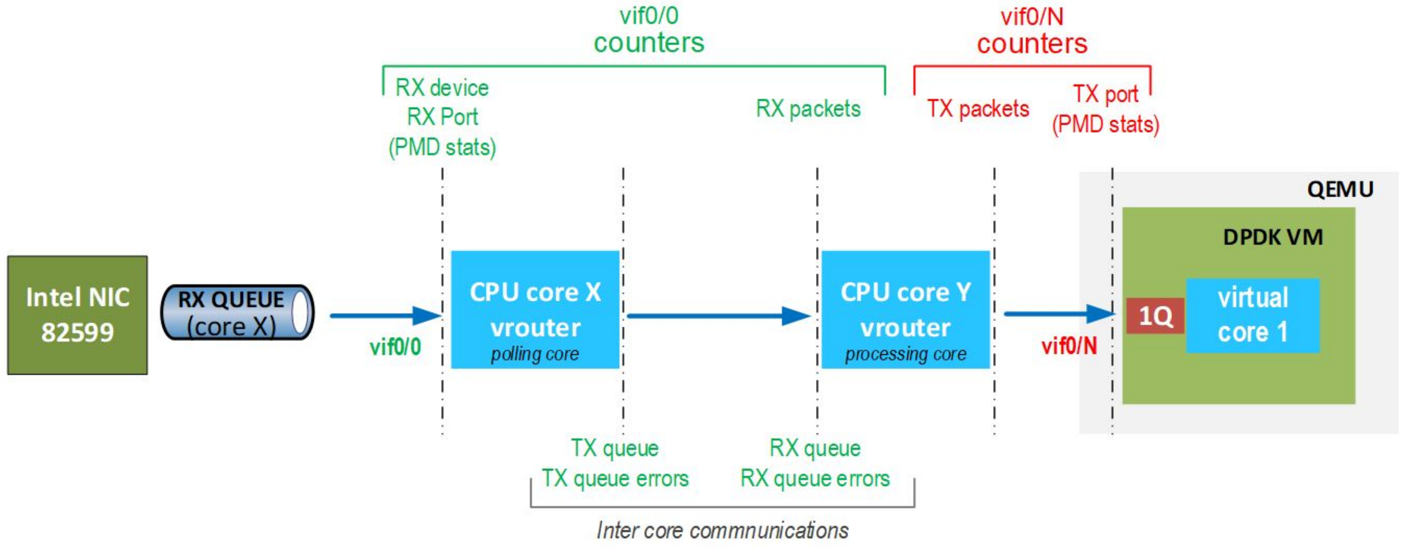
Counter	**Description**
RX device	counter of received packets by physical interface (NIC card), used only for vif 0/0. **Warning!** counters are from vif 0/0 not from physical NIC
RX port	counter of received packets by logical port (aka PMD statistics)
RX queue	counter of inter core communication queues usage
RX queue errors	error counter of lcores queues usage, physical cores used for packets pulling. lcores ID starting from position 10
RX packets	counter of received packets by "processing" core. This is the packets received by vrouter (after passing through NIC, PMD and load-balancing core).
TX device	counter of sent packets by physical interface (NIC card), used only for vif 0/0. **Warning!** counters are from vif 0/0 not from physical NIC.
TX port	counter of sent packets by logical port (aka PMD statistics)
TX queue	counter of inter core communication queues usage
TX queue errors	error counter of lcores queues usage, physical cores used for packets pulling. lcores ID starting from position 10.
TX packets	counter of sent packets by "processing" core.
Drops	counter of drops, to get more info run command with option --get-drop-stats
Syscalls	number of interrupts raised in the guest using eventfd (wait/notify mechanism by user-space applications, and by the kernel to notify user-space applications of events).

Packet flow from a Compute Physical NIC to a VM NIC - counters placement

Compute physical vif is always defined as vif0/0 into the vrouter. vNIC vif are defined as vif0/N (with N above or equal 3).
From underlay to virtual instances, packets are processed into 3 differents steps :

- vif0 packet polling by a polling core
- vif0 packet processing by a forwarding core
- vif0/n packet delivery by a forwarding core

In order to check the 2 first steps, we have to use counters on vif0/0. For the last one, we have to use counters on vif0/n. In the following diagram is shown counters that we have to check on each interface to be able to follow packets processing from vif0/0 to vif0/n into the vrouter :



Here on vif 0/0, RX port and RX packets have the same value :

```
vif0/0      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
            Type:Physical HWaddr:b8:85:84:44:50:83 IPaddr:0.0.0.0
            Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpEr QOS:-1 Ref:147
            RX device packets:11581241437  bytes:4298162385219 errors:0
            RX port   packets:2067782 errors:0
            RX queue  packets:15 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:2067782  bytes:511158595 errors:0
            TX packets:2222204  bytes:654123820 errors:0
            Drops:1037474
            TX port   packets:2228775 errors:20
            TX device packets:4545746  bytes:920251684 errors:0
```

This is the expected situation. It means no packet has been dropped inside the vrouter.

On a receiving VM vif, TX port and TX packets should also have the same value :

```
vif0/10     PMD: tap14cebe1c-c5
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:172.20.10.20
            Vrf:2 Mcast Vrf:2 Flags:L3L2DEr QOS:-1 Ref:19
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:4665  bytes:196158 errors:0
            ISID: 0 Bmac: 02:14:ce:be:1c:c5
            Drops:0
            TX port   packets:1105 errors:3560 syscalls:1105
```

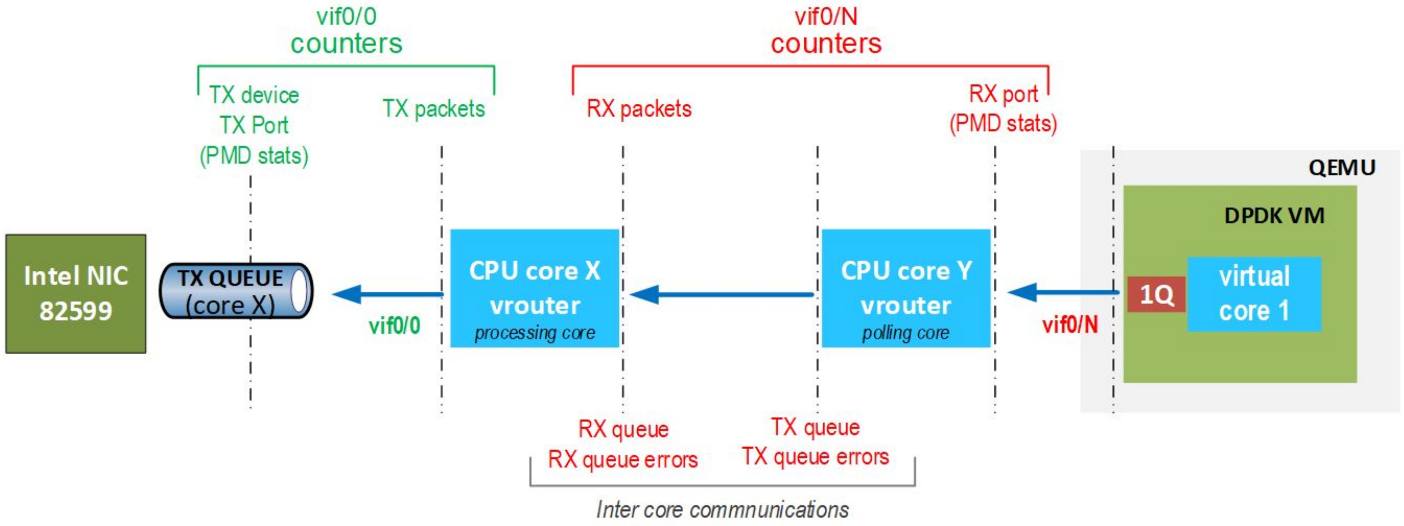
Here 4665 packets have been processed by the “forwarding core”, but only 1105 have been sent to VM interface, 3560 where errored. We can also take note that 1105 “syscalls” notifications have been raised (1 per packet sent to the VM) which shows this instance is not a DPDK one.

This is an unexpected situation. It shows the virtual instance was not able to process all incoming packets in its vif queues and some of them have been dropped.

<u>Packet flow from a VM NIC to compute node Physical NIC - counters placement</u>

Compute physical vif is always defined as vif0/0 into the vrouter. vNIC vif are defined as vif0/N (with N above or equal 3).
From virtual instances to underlay, packets are processed into 3 differents steps :

- vif0/n packet polling by a polling core
- vif0/n packet processing by a forwarding core
- vif0/0 packet delivery by a forwarding core



Here on a transmitting VM vif, RX port and RX packets have the same value :

```
vif0/11      PMD: tapb2767cbd-40
             Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:194.3.0.8
             Vrf:5 Mcast Vrf:5 Flags:L2Er QOS:-1 Ref:17
             RX port   packets:1433409 errors:0 syscalls:2
             RX queue  packets:516225 errors:0
             RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
             RX packets:1433409  bytes:191289533 errors:0
             TX packets:1544574  bytes:374354192 errors:0
             ISID: 0 Bmac: 02:00:00:01:fd:01
             Drops:673575
             TX port   packets:1544236 errors:338 syscalls:1450386
```

This is the expected situation. It means no packet has been dropped inside the vrouter.

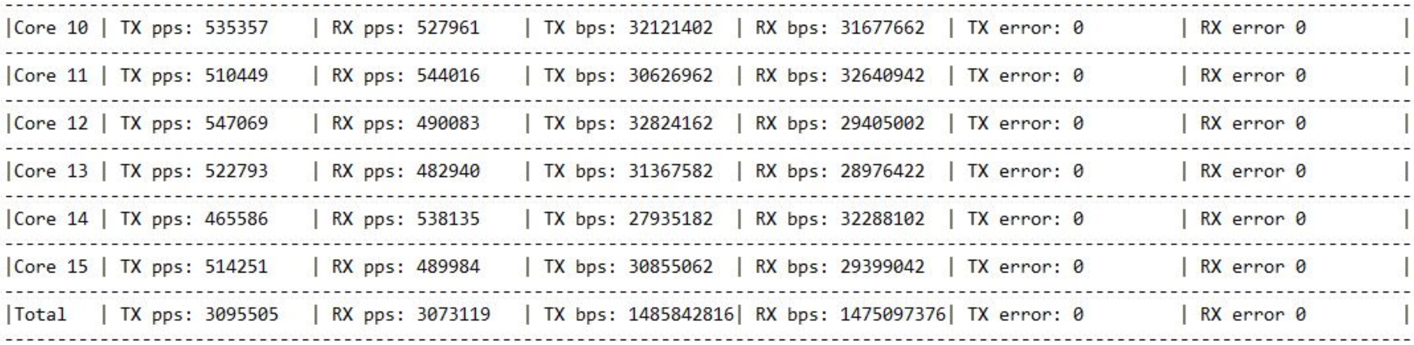
Traffic Load balancing on forwarding cores

A tool for getting statistics per core pps, bw utilisation that can be used for monitoring or testing performance purpose:

https://github.com/PrzemekGrygiel/DPDKstats

This tool is using vif RX and TX packets counters (packet received and transmitted by forwarding cores). In order to check traffic is well balanced on all forwarding cores for a given interface, we have to use following command:
./dpdkvifstats.py -v **<N>**
with **<N>**, interface number to be checked.

For instance:
$ ./dpdkvifstats.py -v **3**



```
------------------------------------------------------------------------------------------------------------------------------
|Core 10 | TX pps: 535357    | RX pps: 527961    | TX bps: 32121402  | RX bps: 31677662  | TX error: 0          | RX error 0          |
------------------------------------------------------------------------------------------------------------------------------
|Core 11 | TX pps: 510449    | RX pps: 544016    | TX bps: 30626962  | RX bps: 32640942  | TX error: 0          | RX error 0          |
------------------------------------------------------------------------------------------------------------------------------
|Core 12 | TX pps: 547069    | RX pps: 490083    | TX bps: 32824162  | RX bps: 29405002  | TX error: 0          | RX error 0          |
------------------------------------------------------------------------------------------------------------------------------
|Core 13 | TX pps: 522793    | RX pps: 482940    | TX bps: 31367582  | RX bps: 28976422  | TX error: 0          | RX error 0          |
------------------------------------------------------------------------------------------------------------------------------
|Core 14 | TX pps: 465586    | RX pps: 538135    | TX bps: 27935182  | RX bps: 32288102  | TX error: 0          | RX error 0          |
------------------------------------------------------------------------------------------------------------------------------
|Core 15 | TX pps: 514251    | RX pps: 489984    | TX bps: 30855062  | RX bps: 29399042  | TX error: 0          | RX error 0          |
------------------------------------------------------------------------------------------------------------------------------
|Total   | TX pps: 3095505   | RX pps: 3073119   | TX bps: 1485842816| RX bps: 1475097376| TX error: 0          | RX error 0          |
------------------------------------------------------------------------------------------------------------------------------
```

Here we can see that the traffic is well balanced on all cores for vif 0/3. Whatever the direction (vrouter to vif 0/3 = TX pps or vif0/3 to vrouter = TX pps), whatever the core, each core is processing around 500KPPS.

vrouter is dropping packets

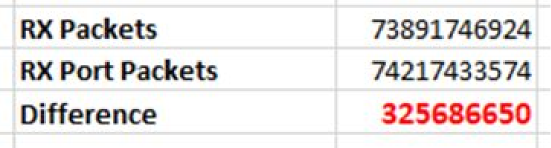
When drops are occurring inside the vrouter, it can be checked into “RX queue errors” counter or “IF Drop” in drop statistics option.

```
$ vif --get 3 --get-drop-stats |  grep "IF Drop“
IF Drop                          325686650
```

```
$ vif --get 3 --get-drop-stats |  grep "RX queue  packets“
            RX queue  packets:73891601519 errors:325686650
```

This value also matches RX Ports packets – RX Packets value:

```
$ vif --get 3 --get-drop-stats |  grep "RX packets"
        RX packets:73891746924  bytes:4433501752754 errors:0

$ vif --get 3 --get-drop-stats |  grep "RX port"
        RX port   packets:74217433574 errors:0 syscalls:1
```



RX Packets	73891746924
RX Port Packets	74217433574
Difference	**325686650**

This is a situation that is occuring when too many packets are sent onto one or more processing cores. It means :

- traffic is not equally load balanced on all forwarding cores
- or not enough all forwarding cores have been allocated to the vrouter to be able to process the incoming traffic.

We can check on which forwarding core the drops have been occured using following command:

```
$ vif --get <interface> --core <forwarding core>  grep "RX queue errors"
```

For instance:

```
$ vif --get 3 --core 11 | grep "RX queue errors"
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 27283689 0 7534963 9294274 6851459 5748667
```

xxx: forwarding core which drops packets

xxx: polling cores whose packets have been dropped

Currently (19.12 release with an optimal setup), a forwarding core is able to process around 500 KPPS in each direction (RX and TX), so 1 MPPS in total for 64 bytes packets.

virtual instance is dropping packets

When drops are occurring because a virtual instance is not fast enough to process incoming packet, it can be detected using TX port errors counters :

```
$ vif --get 10 | grep “TX port \|TX packets”
            TX packets:4665  bytes:196158 errors:0
            TX port   packets:1105 errors:3560 syscalls:1105
```

Here we are detecting that 3560 packets have been dropped on TX port. It means VM was to slow to process incoming packets. Some of them have been dropped into the virtio queues.

Packet Drop statistics

With dropstats command we can get details about drops performed by the vrouter.
We can get all parameters supported by this command using :

```
$ dropstats --help
```

When some packet loss issue is encountered, this is a good starting point to get an overview of all drops that have been performed on the vrouter.

dropstats command run without any arguments is returning drop statistics for all vrouter cores :

```
$ dropstats
...
```

In order to get statistics for a given vrouter core, we can use the --core option. Keep in mind that 10 is the first processing vrouter lcore:

```
$ dropstats --core 10
...
```

In order to clear stats counters on all cores we can use this command :

```
$ dropstats --clear
```

<u>Example</u>:

```
$ (vrouter-agent-dpdk)[root@overcloud-contraildpdk-1 /]$ dropstats
Invalid IF                    0
Trap No IF                    0
IF TX Discard                 0
IF Drop                       0
IF RX Discard                 0

Flow Unusable                 0
Flow No Memory                0
Flow Table Full               0
Flow NAT no rflow             0
Flow Action Drop              0
Flow Action Invalid           0
Flow Invalid Protocol         0
Flow Queue Limit Exceeded     0
New Flow Drops                0
Flow Unusable (Eviction)      0

Original Packet Trapped       0

Discards                      0
TTL Exceeded                  0
Mcast Clone Fail              0
Cloned Original               0
```

```
Invalid NH                    27
Invalid Label                 0
Invalid Protocol              0
Etree Leaf to Leaf            0
Bmac/ISID Mismatch            0
Rewrite Fail                  0
Invalid Mcast Source          0
Packet Loop                   0

Push Fails                    0
Pull Fails                    0
Duplicated                    0
Head Alloc Fails              0
PCOW fails                    0
Invalid Packets               0

Misc                          0
Nowhere to go                 0
Checksum errors               0
No Fmd                        0
Invalid VNID                  0
Fragment errors               0
Invalid Source                0
Jumbo Mcast Pkt with DF Bit   0
No L2 Route                   0
Memory Failures               0
Fragment Queueing Failures    0
No Encrypt Path Failures      0

VLAN fwd intf failed TX       0
VLAN fwd intf failed enq      0
```

<u>dropstats ARP Block</u>

GARP	GARP packets from VMs are dropped by vrouter.This is an expected behavior. This counter indicates how many GARP packets were dropped.
ARP notme	this counter is incremented when ARP requests that are not handled by vrouter are dropped. For example, requests for a system that is not a host. These drops are counted by counters.
Invalid ARPs	this counter is incremented when the Ethernet protocol is ARP, but the ARP operation was neither a request nor a response.

dropstats Interface Block

Invalid IF	this counter is incremented normally during transient conditions, and should not be a concern.
Trap No IF	this counter is incremented when vrouter is not able to find the interface to trap the packets to vrouter agent, and should not happen in a working system
IF TX Discard and IF RX Discard	these counters are incremented when vrouter is not in a state to transmit and receive packets, and typically happens when vrouter goes through a reset state or when the module is unloaded.
IF Drop	this counter indicates packets that are dropped in the interface layer. The increase can typically happen when interface settings are wrong or vrouter was not powerful enough to process all incoming packets.

dropstats Flow Block

Flow unusable counter	When packets go through flow processing, the first packet in a flow is cached and the vrouter agent is notified so it can take actions on the packet according to the policies configured. If more packets arrive after the first packet but before the agent makes a decision on the first packet, then those new packets are dropped. The dropped packets are tracked by this counter
Flow No Memory	this counter increments when the flow block doesn't have enough memory to perform internal operations. The Flow Table Full counter increments when the vrouter cannot install a new flow due to lack of available slots. A particular flow can only go in certain slots, and if all those slots are occupied, packets are dropped. It is possible that the flow table is not full, but the counter might increment.
Flow NAT no rflow	this counter tracks packets that are dropped when there is no reverse flow associated with a forward flow that had action set as NAT. For NAT, the vrouter needs both forward and reverse flows to be set properly. If they are not set, packets are dropped.
Flow Action Drop	this counter tracks packets that are dropped due to policies that prohibit a flow.
Flow Action Invalid counter	this counter usually does not increment in the normal course of time, and can be ignored.

Flow Invalid Protocol	this counter usually does not increment in the normal course of time, and can be ignored.
Flow Queue Limit Exceeded	More than three packets received before flow processing is completed. This counter usually does not increment in the normal course of time, and can be ignored.

dropstats Miscellaneous Operational Block

Discard	this counter tracks packets that hit a discard next hop. For various reasons interpreted by the agent and during some transient conditions, a route can point to a discard next hop. When packets hit that route, they are dropped.
TTL Exceeded	this counter increments when the MPLS time-to-live goes to zero.
Mcast Clone Fail	it happens when the vrouter is not able to replicate a packet for flooding.
Cloned Original	Original packet is dropped, after action is taken on cloned packet. This is an internal tracking counter. **It is harmless and can be ignored.**
Invalid NH	this counter tracks the number of packets that hit a next hop that was not in a state to be used (usually in transient conditions) or a next hop that was not expected, or no next hops when there was a next hop expected. Such increments happen rarely, and **should not continuously increment.**
Invalid Label	this counter tracks packets with an MPLS label unusable by vrouter because the value is not in the expected range.
Invalid Protocol	typically increments when the IP header is corrupt.
Rewrite Fail	this counter tracks the number of times vrouter was not able to write next hop rewrite data to the packet.
Invalid Mcast Source	this counter tracks the multicast packets that came from an unknown or unexpected source and thus were dropped.
Invalid Source	this counter tracks the number of packets that came from an invalid or unexpected source and thus were dropped (RPF check on the packet failed).

Other counters (mainly intended for developers)

Push Fails	Increasing the length of buffer failed
Pull Fails	Decreasing the length of buffer failed
Duplicated	Trap packet to Agent has been truncated and original packet is dropped
PCOW fails	Cloning and copying of buffer failed
Invalid Packets	Format of the received packet is Invalid
Misc	Packet dropped for miscellaneous reasons

Nowhere to go	Packet dropped because intended L3 or L2 processing failed
Checksum errors	Packet dropped because of Checksum failure
No Fmd	No forwarding metadata to process the packet
Invalid VNID	Wrong VNI/Vxlan Id received in the packet
Fragment errors	Enqueueing to fragment queue failed
Jumbo Mcast Pkt with DF Bit	Multicast packet more than MTU size is received
No L2 Route	L2 route is missing

vif command: get-drop-stats option

In order to get drops statistics generated on a given vrouter interface we can use this command :

$ vif --get ***<vif ID>*** --get-drop-stats

Adding --core option we can get these statistics for a given vrouter lcore :

$ vif --get ***<vif ID>*** --get-drop-stats --core ***<core ID>***

As for dropstats command, let’s keep in mind that 10 is the first processing vrouter lcore:

<u>Example</u>:

$ vif --get 0 --get-drop-stats

```
Vrouter Interface Table

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
       Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer
2
       D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
       Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering
Offload, Mon=Interface is Monitored
       Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New
Flows, L=MAC Learning Enabled
       Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without
Vlan Tag, Ig=Igmp Trap Enabled

vif0/0      PCI: 0000:00:04.0 (Speed 1000, Duplex 1)
            Type:Physical HWaddr:52:54:00:f2:9f:f6 IPaddr:0.0.0.0
            Vrf:0 Mcast Vrf:65535 Flags:L3L2 QOS:0 Ref:20
            RX device packets:1021114  bytes:0 errors:0
            RX port   packets:1021114 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
            RX packets:1021114  bytes:157010437 errors:0
            TX packets:1063497  bytes:690713560 errors:0
            Drops:0
            TX queue  packets:104822 errors:0
            TX port   packets:1063497 errors:0
            TX device packets:1063497  bytes:0 errors:0

Invalid IF                    0
Trap No IF                    0
IF TX Discard                 0
IF Drop                       0
IF RX Discard                 0
...
```

Dropstats and packet drop log utility

Packet Drop log utility is an extension of dropstats command (available since Contrail 5.0) which logs drop packet information like : timestamp, drop reason, vif_index, nexthop id, packet type, source IP, destination IP, source port, destination port, packet length, packet data, filename and line number.

Packet drop log utility is enabled by default with minimum logging (log only timestamp, drop reason, file name and line number).

In further release, IP source and destination will be added to this minimum logging information.

vrouter kernel version

Packet drop log can be enabled/disabled and its buffer size can be changed during load time by setting some options in /etc/modprobe.d/vrouter.conf :

```
options vrouter vr_pkt_droplog_bufsz=<value>
options vrouter vr_pkt_droplog_buf_en=1
```

Packet drop log and minimum logging feature during can be enabled/disabled during runtime using following command:

```
$ echo 1 > /proc/sys/net/vrouter/pkt_drop_log_enable
$ echo 1 > /proc/sys/net/vrouter/pkt_drop_log_min_enable
```

vrouter DPDK version

Currently neither *vrouter vr_pkt_droplog_buf_en* nor *vr_pkt_droplog_bufsz* parameters can be modified when using a DPDK vrouter version.This is also not possible to disable packet drop log nor minimum logging feature on a dpdk vrouter.

PS: The defined buffer size is the number of entries available for each processing lcore. The default value is set to 200 entries. On a DPDK vrouter which is using 4 lcore, we can collect up to 4x200 = 800 dropped packets

Command:

5.1 release:

```
$ dropstat -log 0
```

display log packet drops for all cores.

```
$ dropstats -log <core number>
```

display log packet drops for the given core.

Full log table is displayed by this command, so empty lines have to be filtered out:

```
sl no: 0  Epoch Time: 0 Local Time: Thu Jan  1 00:00:00 1970
 Packet Type: 0  Drop reason: NULL  Vif idx: 0  Nexthop id: 0  Src IP: NULL  Dst IP: NULL  Source port: 0  Dest port: 0  file: NULL
line no: 0  Packet Length: 0  Packet Data:
```

It can be done using a filter onto the Drop reason :

```
(vrouter-agent-dpdk)[root@overcloud-contraildpdk-1 /]$ dropstats -l 0 | grep -C3  INVALID_NH | head -n 8
Pkt Drop Log for Core 11

sl no: 0  Epoch Time: 1569247326 Local Time: Mon Sep 23 14:02:06 2019
 Packet Type: 0  Drop reason: VP_DROP_INVALID_NH  Vif idx: 0  Nexthop id: 0  Src IP: NULL  Dst IP: NULL  Source port: 0  Dest
port: 0  file: VR_NEXTHOP_C  line no: 2656  Packet Length: 0  Packet Data:

sl no: 1  Epoch Time: 1569247326 Local Time: Mon Sep 23 14:02:06 2019
 Packet Type: 0  Drop reason: VP_DROP_INVALID_NH  Vif idx: 0  Nexthop id: 0  Src IP: NULL  Dst IP: NULL  Source port: 0  Dest
port: 0  file: VR_NEXTHOP_C  line no: 2656  Packet Length: 0  Packet Data:
```

19.7 release and above:

```
$ dropstat --log 0
```

display log packet drops for all cores.

```
$ dropstats --log <core number>
```

display log packet drops for the given core.

PS: with --log option, the lcore number to be given is the real coreID + 1.

Workaround - How to disable minimum logging feature on Contrail DPDK vrouter :

Currently there is no simple way to disable minimum logging feature onto a Contrail DPDK vrouter. In this section we are describing a workaround that could be used onto some development environment in order to collect full packet drop logs onto Contrail DPDK vrouter. **This workaround must not be used onto a production environment**.

1 - download contrail-debug container from the Juniper Docker Hub:

```
$ sudo docker pull hub.juniper.net/contrail-debug:<tag>
```

<tag> is depending on the release used, 5.1.0-0.38-rhel-queens for instance.

2 - copy contrail-vrouter-dpdk.debug fileinside DPDK container at /usr/bin

```
$ sudo -i
# cd /var/lib/docker/overlay2
# find . -name contrail-vrouter-agent.debug
# find . -name contrail-vrouter-dpdk.debug
```

Keep note the path of the last command :

```
# docker cp <contrail-vrouter-dpdk.debug path>/contrail-vrouter-dpdk.debug   <contrail-vrouter-dpdk name or ID>:/usr/bin
```

Here for instance, our dpdk vrouter agent container name is contrail-vrouter-agent-dpdk and its ID: 3920ca0aac69

```
# docker ps | grep  vrouter | grep dpdk
3920ca0aac69      192.168.24.1:8787/contrail/contrail-vrouter-agent-dpdk:1909.30-rhel-queens   "/entrypoint.sh /u..."   7 hours ago
Up 14 minutes                         contrail-vrouter-agent-dpdk
```

3 - run gdb on to contrail vrouter dpdk binary

```
# docker exec -it <contrail-vrouter-dpdk name or ID>  bash
(container)#  ps -eaf | grep contrail-vrouter-dpdk
```

Keep note the process ID: <DPDK PID>

```
(container)# gdb /usr/bin/contrail-vrouter-dpdk
```

GDB will load both *contrail-vrouter-dpdk* and *contrail-vrouter-dpdk.debug* binary files.

4 - attach to the current running dpdk process

```
(gdb) attach <DPDK PID>
```

5 - Print and update the variable vr_pkt_droplog_min_sysctl_en

```
(gdb) p vr_pkt_droplog_min_sysctl_en
(gdb) set vr_pkt_droplog_min_sysctl_en=0
```

Faulty vrouter deployment troubleshooting

Some typical root cause of a faulty OpenStack contrail deployment failure is a network misconfiguration onto compute nodes. Openstack is using *os-net-config* to perform network configuration onto network hosts.

os-net-config CLI is using YAML or JSON file formats. By default os-net-config uses a YAML config file located at /etc/os-net-config/config.yaml. It can be customized via the --config-file CLI option.

Providers are used to apply the desired configuration on the host system. By default 3 providers are implemented:

- Ifcfg: persistent network config format stored in /etc/sysconfig/network-scripts
- ENI: persistent network config format stored in /etc/network/interfaces
- iproute2: non-persistent provider which implements the config using iproute2, vconfig, etc...

When using bin/os-net-config the provider is automatically selected based on the host systems preferred persistent network type (ifcfg or ENI). This can be customized via the --provider CLI option.

When vrouter is failing to start onto a compute node we have first to check the configuration that has been tried to be applied by the deployment tool (Ansible, TripleO for instance). It can be checked using the following command :

```
$ cat /etc/os-net-config/config.json | jq.
```

vhost0 (vrouter main interface) activation failure can be checked with the following one:

```
$ sudo journalctl -u os-collect-config
```

os-net-config CLI tools can also be run manually in order to try to find the root cause of a faulty vrouter deployment. For instance following command can be launched onto a faulty compute node :

```
$ sudo os-net-config –c /etc/os-net-config/config.json
```

Contrail vrouter logs have also to be checked. For a DPDK vrouter it can be checked into /var/log/contrail/contrail-vrouter-dpdk.log (up to 4.1) or /var/log/containers/contrail/contrail-vrouter-dpdk.log (from 5.0 and above).

```
$ cat /var/log/containers/contrail/contrail-vrouter-dpdk.log
```

When DPDK is used, it’s also important to check that some DPDK interfaces are available to be used by the vrouter. It can be checked with :

```
$ sudo /opt/contrail/bin/dpdk_nic_bind.py -s
```

References

Some more information about performance and DPDK :

https://developers.redhat.com/blog/2017/06/28/ovs-dpdk-parameters-dealing-with-multi-numa/
https://doc.dpdk.org/guides/linux_gsg/nic_perf_intel_platform.html
https://software.intel.com/en-us/articles/dpdk-performance-optimization-guidelines-white-paper
https://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html

Some reference about the kernel args:

https://access.redhat.com/articles/3720611
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html-single/tuning_guide/index#Isolating_CPUs_Using_tuned-profiles-realtime