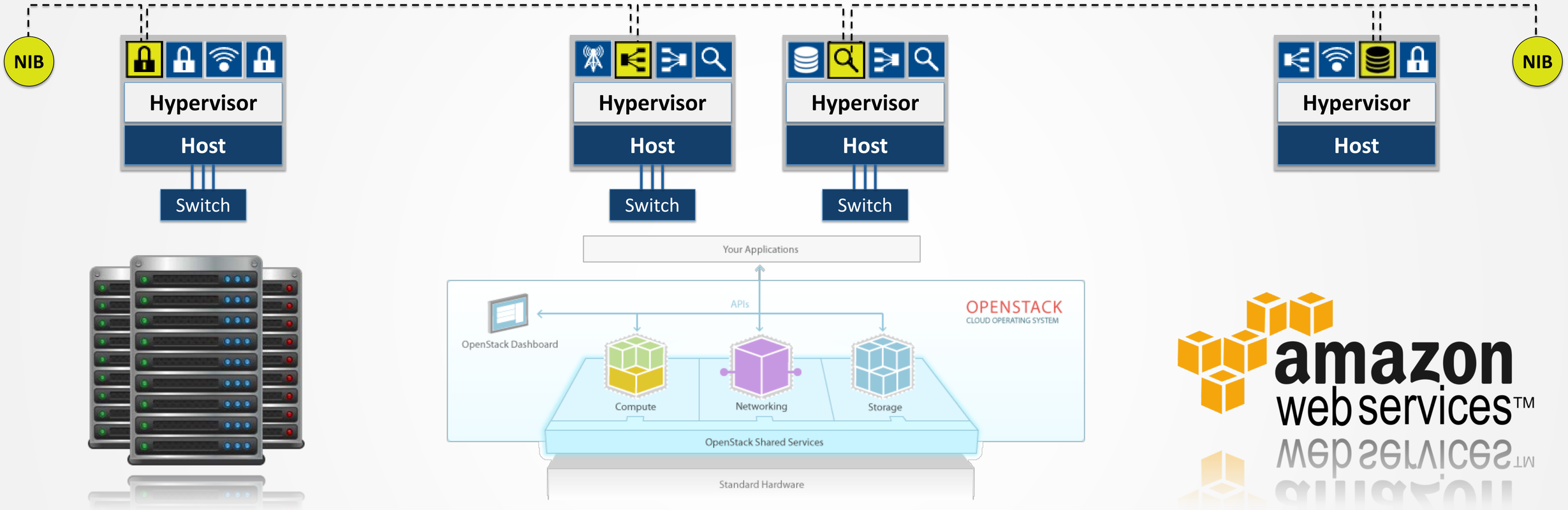
RIFT.io

Leveraging DPDK to Scale-Out Network Functions Without Sacrificing Networking Performance

August 2015

VNFs Will Run in Diverse Infrastructures



Bare Metal

Private Clouds

Public Clouds

How can we unify these environments?

Virtual Network Function (VNF) Considerations

Example VNF

A VNF Breaks Down Into VNF Components (VNFc)

- Control Plane / Data Plane / Security as example
- VNF vendors likely to have diverse network attachment models
- Any NFV architecture needs to accommodate a variety of guest attachment options
- Each VNFC will need to scale independently and be able to run virtually anywhere

VNFc-1

- Packet forwarding performance is the most critical metric
- Limit complex interface bonding if possible for flow state consistency
- Guest OS's need access to data plane accelerated NICs (DPDK)
- Target hosts with high bandwidth NICs (40G/100G)

VNFc-2

- Transactions per second is the most critical metric
- VPN / NAT association and management
- Many other in-line services likely to be offered
- Responsible for coordinated scale-out of all VNFCs
- Likely most utilized function – target low cost hosts

VNFc-3

- Session encryption / decryption rate is the most critical metric
- Scale-out of crypto tunnels across security subsystem key for deterministic scalability
- Guest OS's need access to crypto offload
- Target hosts with PCI accelerated crypto processing (NPU/NSP)

Variety of Network Attachment Options

Virtio

- Para-virtualized network that's the simplest to deploy
- OpenStack native support
- Tenant encapsulation supported by OpenStack
- Lower performance due to many context switches (host / guest / QEMU)
- Complex networking limited to host environment

Direct Pass-Through

- Direct guest access to NIC hardware
- OpenStack does not natively support this
- Tenant encapsulation outside of OpenStack - significant work to integrate
- Very high performance due to direct guest access to the hardware
- Complex networking left to the guest environment and underlay

SR-IOV

- High speed multi-guest NIC access
- OpenStack native support
- Tenant encapsulation outside of OpenStack - significant work to integrate
- High performance due to direct hardware support
- Complex networking left to the guest environment and underlay

DPDK Accelerated vSwitch with Sockets / KNI

- KNI provides the ability to use guest kernel NIC interfaces
- Supported by Openstack
- Low performance due to kernel interface
- Complex networking limited to host environment

DPDK Accelerated vSwitch with Ivshmem

- Facilitates fast zero-copy data sharing among virtual machines
- Supported by Openstack
- Good performance but hugepage shared by all guests (unsecure)
- Complex networking limited to host environment

DPDK Accelerated vSwitch with Virtio

- Virtio to DPDK in QEMU
- Supported by Openstack
- Limited performance due to packet copy
- Complex networking limited to host environment

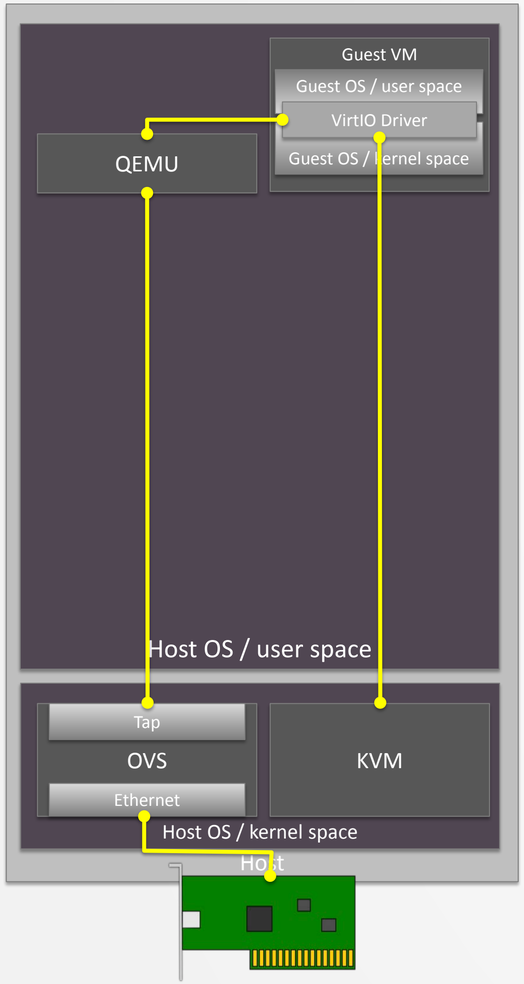
DPDK Accelerated vSwitch with vhost-user

- Facilitates fast zero-copy data sharing among virtual machines
- Supported by Openstack
- Limited performance due single queue limitation (multi-queue coming)
- Complex networking limited to host environment

Network Attachment Option - Details

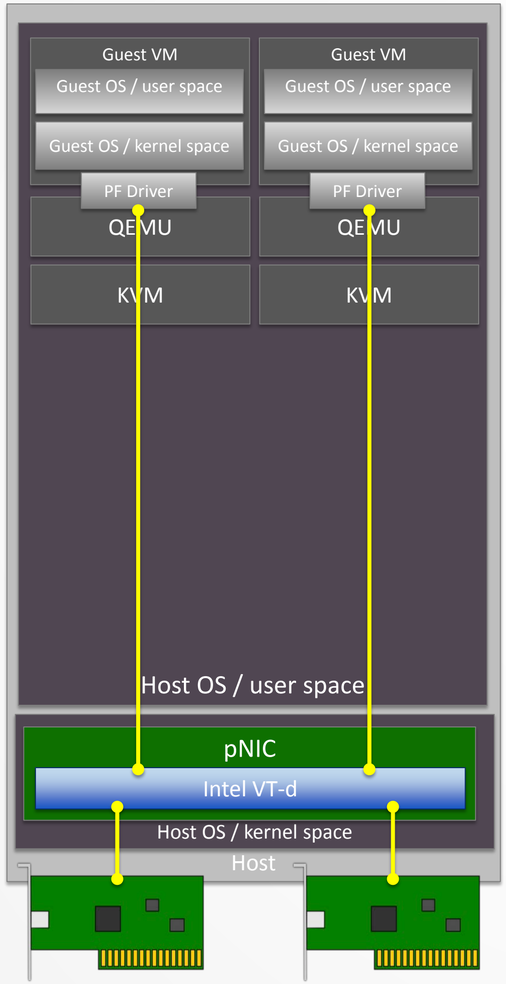
Standard vSwitch

virtIO

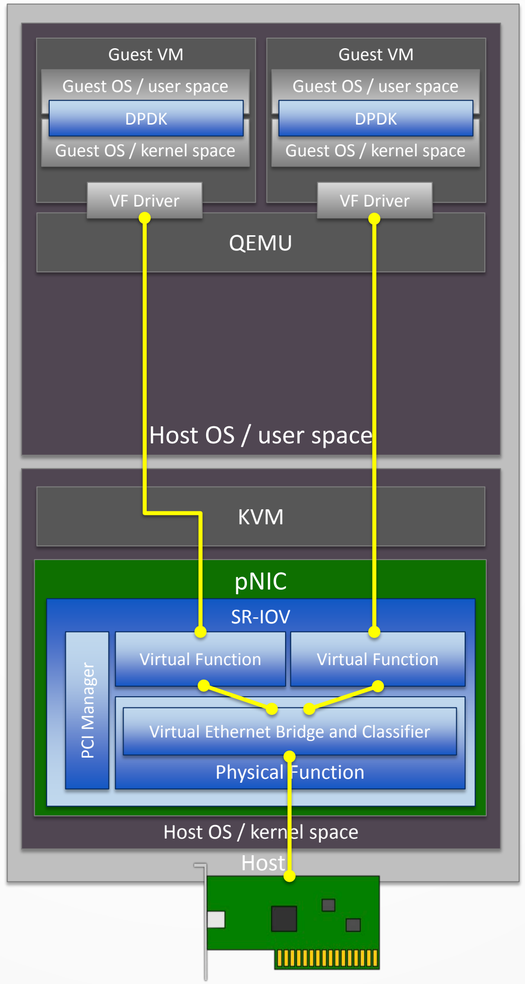


Hypervisor Bypass

Direct Pass-through



DPDK SR-IOV



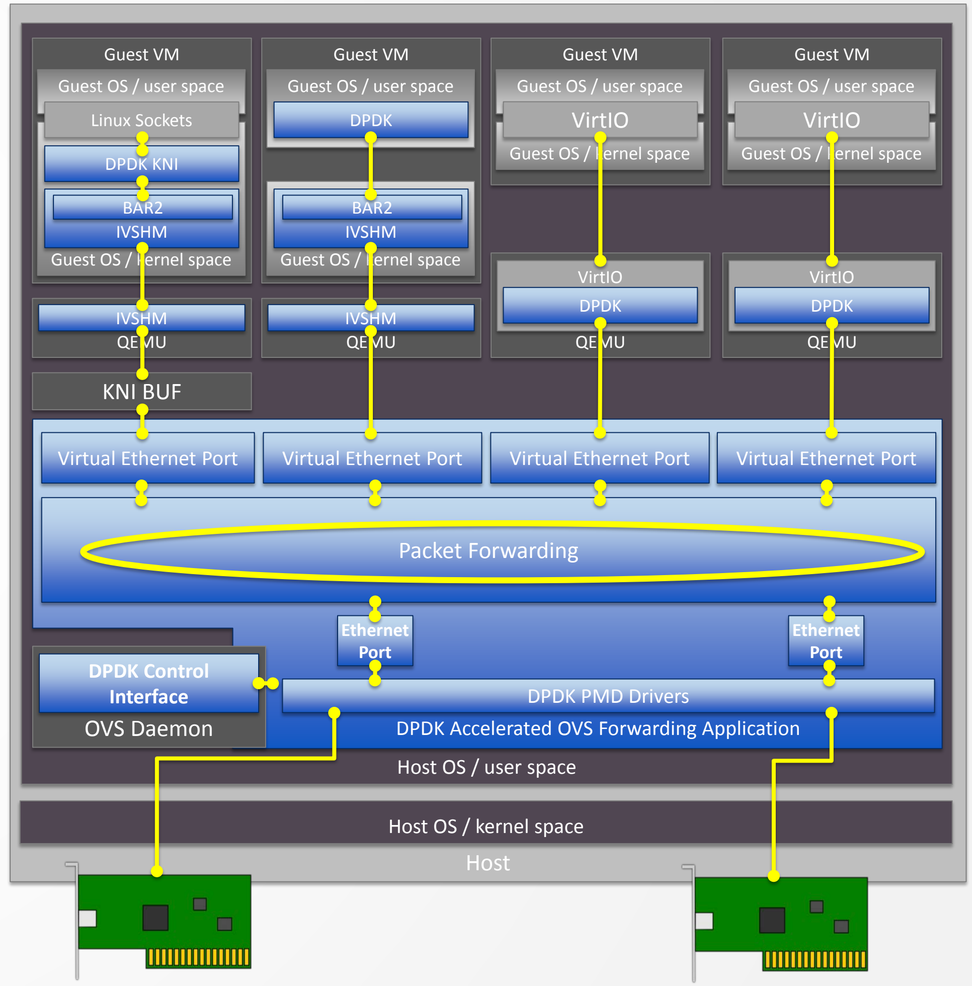
DPDK Accelerated vSwitch

KNI

ivshmem

virtIO

vhost-user



Other Considerations Beyond Network Attachment Options

CPU Pinning

- A process or thread affinity configured with one or multiple cores
- In a 1:1 pinning configuration between virtual CPUs and physical CPUs, some predictability is introduced into the system by preventing host and guest schedulers from moving workloads around facilitating other efficiencies such as improved cache hit rates

Huge Pages

- Provides up to 1-GB page table entry sizes to reduce I/O translation look-aside buffer (IOTLB) misses which improves networking performance, particularly for small packets

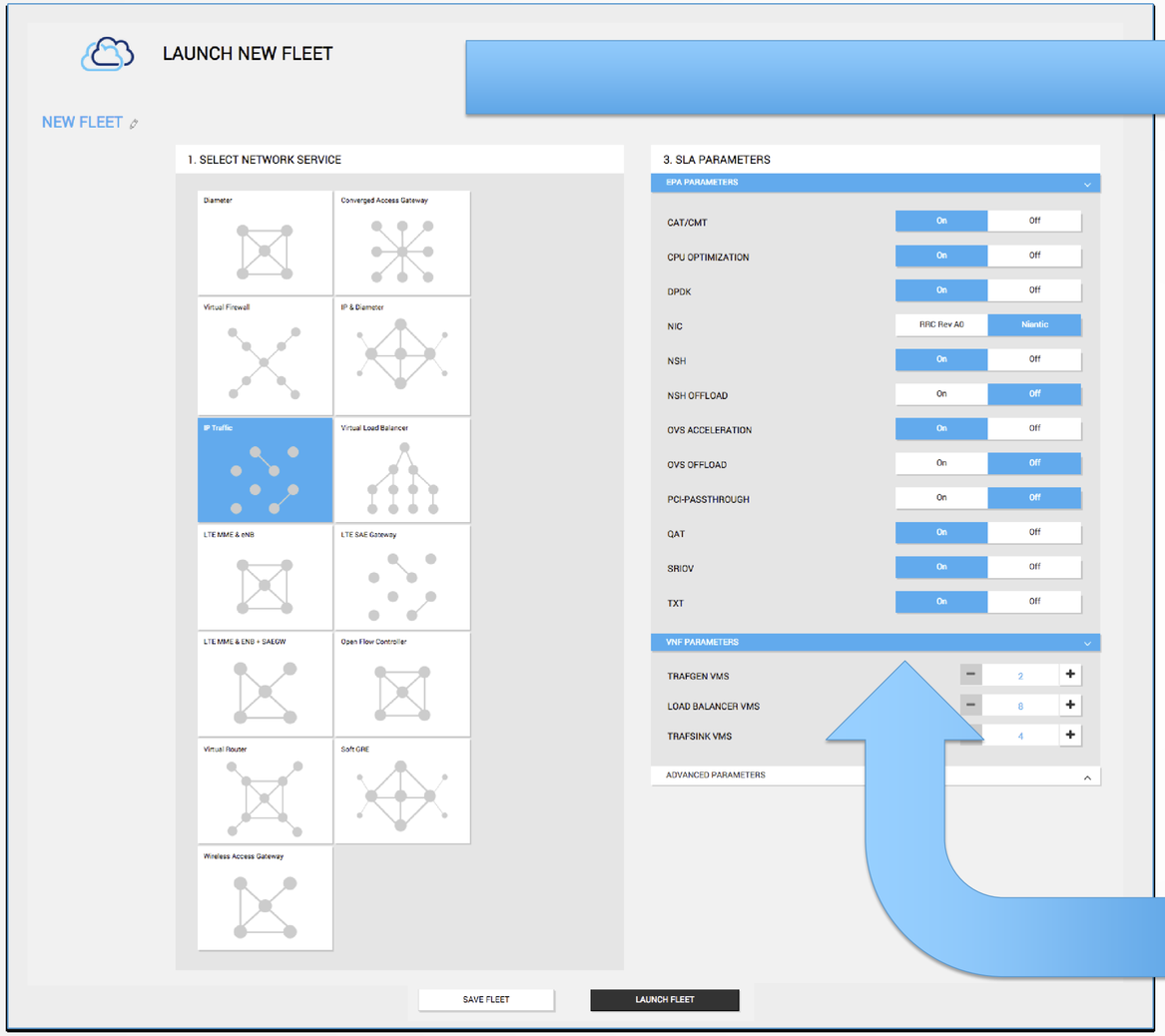
I/O-Aware NUMA Scheduling

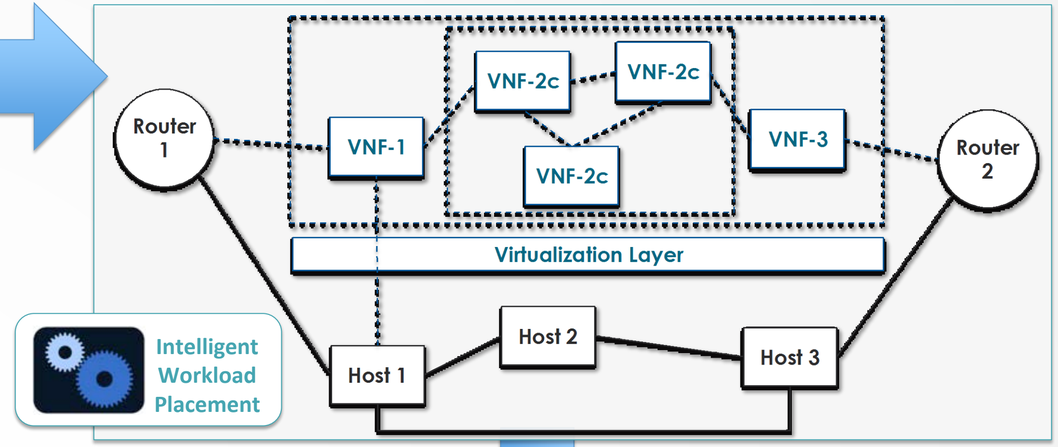
- Memory allocation process that prioritizes the highest-performing memory local to a processor core
- Able to configure VMs to use CPU cores from the same processor socket and choose the optimal socket based on the locality of the relevant NIC device that is providing the data connectivity for the VM

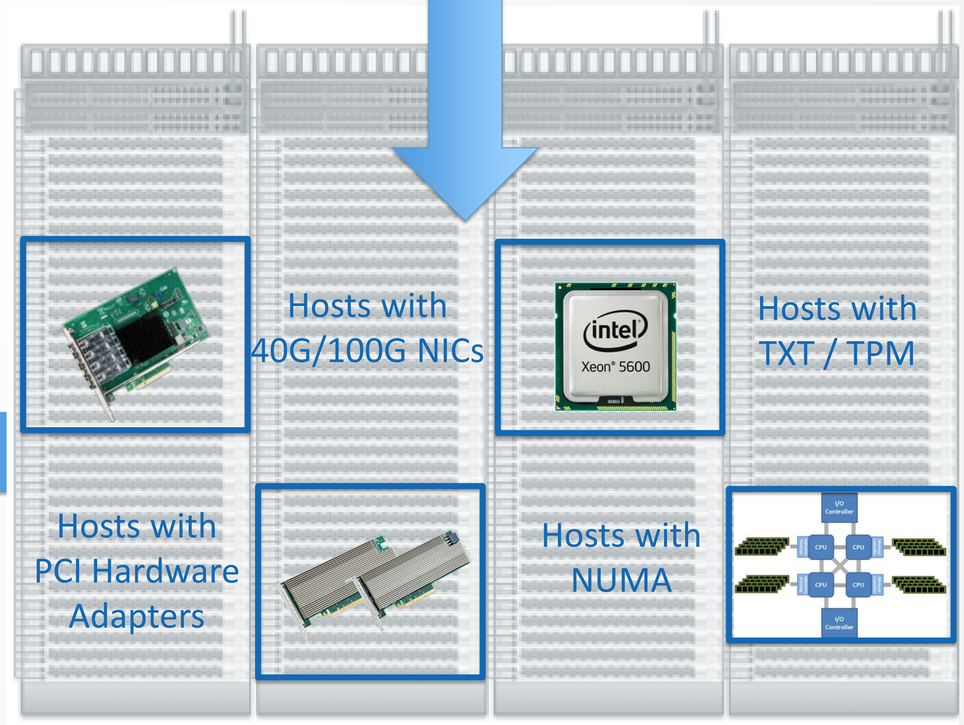
Cache Monitoring Technology / Cache Allocation Technology (CMT/CAT)

- CMT allows an operating system (OS) or hypervisor or virtual machine monitor (VMM) to determine the usage of cache by applications running on the platform
- CMT allows an OS or VMM to assign an ID (RMID) for each of the applications or VMs that are scheduled to run on a core, monitor cache occupancy on a per-RMID basis, and read last level cache occupancy for a given RMID at any time
- CAT allows an OS, hypervisor, or VMM to control allocation of a CPU's shared last-level cache which lets the processor provide access to portions of the cache according to the established class of service (COS)
  - Configuring COS defines the amount of resources (cache space) available at the logical processor level and associates each logical processor with an available COS
- CMT provides an application the ability to run on a logical processor that uses the desired COS

Workload Placement Implications



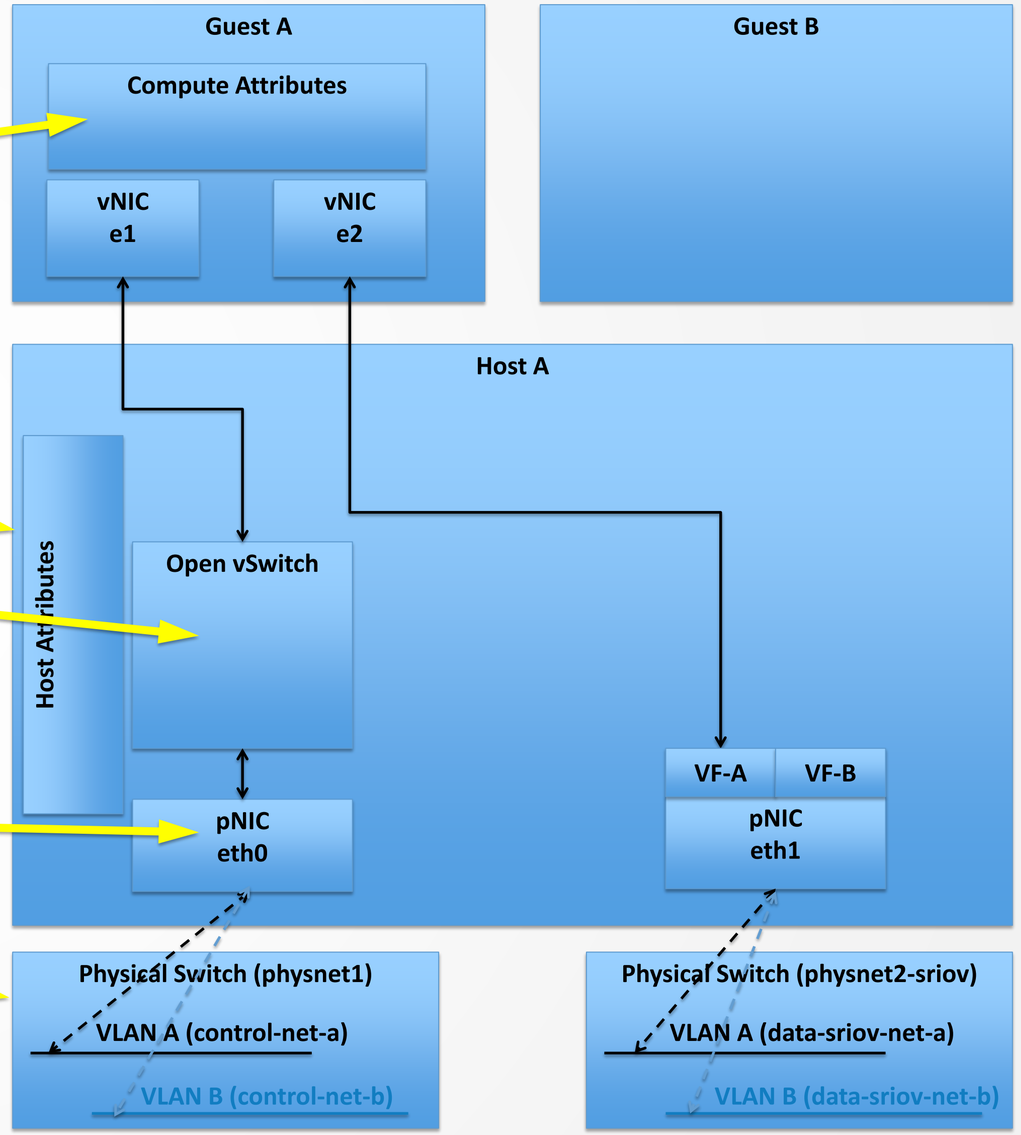




Matching Workload Needs with Infrastructure Capabilities

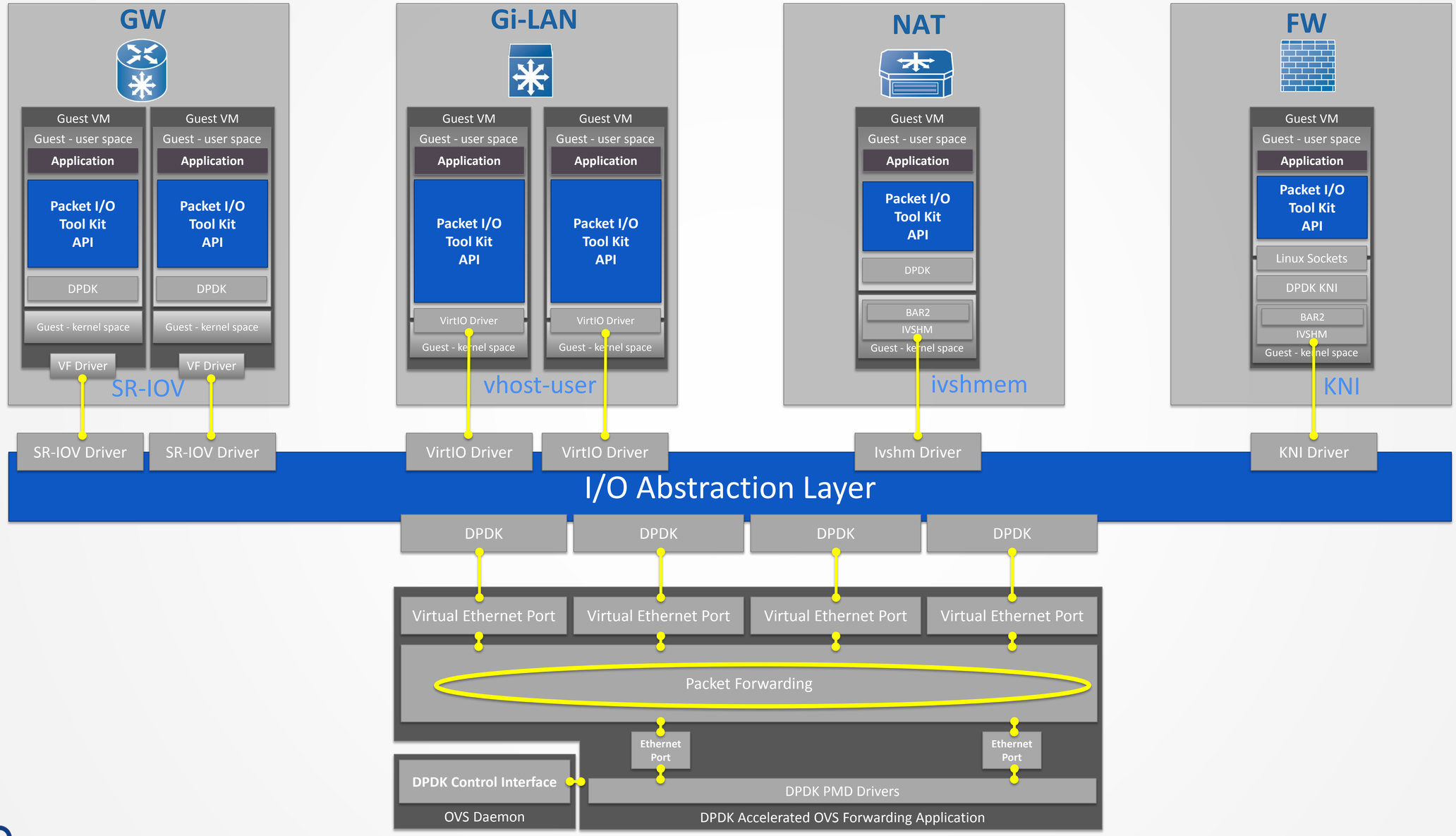
VDU Descriptor:

- Image: <path>
- Flavor: { vcpus: <count>, memory: <mb>, storage: <GB> }
- Guest- EPA: { mempage-size: <large|small|prefer-large>,
  trusted-execution: <true|false>,
  cpu-pinning-policy: <dedicated|shared|any>,
  thread-pin-policy: <avoid|separate|isolated|any> ,
  numa: { node-cnt: <count>,
  mem-policy: <strict|preferred> ,
  nodes: { id: <id>, memory: <mb>,
  vcpu-list: <comma separated list> } } }
- Host-EPA: { processor: { model: <model>,
  features: <64b, iommu, cat, cmt, ddio, etc>
  } }
- vSwitch-EPA: { ovs-acceleration: <true|false>,
  ovs-offload: <true|false> } }
- Hypervisor: { type:<kvm|xen> , version: <> }
- Interface:
  - Name: <string>
  - Type: direct-sr-iov | normal | direct-pci-passthrough
  - NIC-EPA: { vlan-tag-stripping: <boolean>,
    ovs-offload: <boolean>, vendor-id: <vendor-id>,
    datapath-library: <name-of-library>,
    bandwidth: <bw> }
  - Network: <name-of-provider-network>



Detailed CPU and network controls described in an open descriptor model!

The Need for Abstracted I/O



Creating a Packet I/O Toolkit Leveraging DPDK

Device Open - This API is used for opening a PIOT-managed device for I/O

- Input parameters:
  - Device name
  - Number of device Tx queues requested
  - Number of device Rx queues requested
  - Device event callback (link up, link down, etc.)
  - Initial device configuration requested:
  - Promiscuous mode
  - Multicast
- Output:
  - Handle for the opened device, with the following information:
  - Number of Tx queues allocated
  - Number of Rx queues allocated
  - NUMA socket affinity
  - Interrupt event poll info:
  - Event poll function pointer
  - Event poll file descriptor (of /dev/uioN device)

Device Close - This API is used to close the PIOT connection for the device specified by the input handle.

- Input parameters:
  - Device handle
- Output:
  - Success/failure status

Burst Transmit - This API polls the specified receive queue of the device for packets and, if they are available, reads and returns the packets in bulk. The caller can specify the maximum number of packets that can be read.

- Input parameters:
  - Device handle
  - Transmit queue ID
  - Number of packets to be transmitted
  - Packets to be transmitted
- Output:
  - Number of packets transmitted

Burst Receive - This API polls the specified receive queue of the device for packets and, if they are available, reads and returns the packets in bulk. The caller can specify the maximum number of packets that can be read.

- Input parameters:
  - Device handle
  - Receive queue ID
  - Maximum number of packets to be read
- Output:
  - Number of packets received
  - Packets received

Device Start - This API call is used for device-specific start operation.

- Input parameters:
  - Device handle
- Output:
  - Success/failure status

Device Pairing - This operation is applicable only for certain types of devices. Initially, this will be implemented only for Ring Mode devices. Its purpose is to pair two specified logical devices. It works by connecting the receive of one device to the transmit of the other device, and vice-versa, creating a loop back between them.

- Input parameters:
  - Device 1 handle
  - Device 2 handle
- Output:
  - Status

Device Unpairing - This API call is used to unpair paired devices. It is important to note that paired devices must be unpaired before either one can be closed.

- Input parameters:
  - Device 1 handle
  - Device 2 handle
- Output:
  - Status

Device Stop - This API call is used for device-specific start operation.

- Input parameters:
  - Device handle
- Output:
  - Success/failure status

Device Information Fetch - This API function is used to fetch device status and device-specific information.

- Input parameters:
  - Device handle
- Output:
  - Device information:
  - Device driver name
  - Max Rx queues
  - Max Tx queues
  - Max MAC address
  - PCI ID and address
  - NUMA node

Device Statistics Fetch - This API call is used to fetch input and output statistics for the device.

- Input parameters:
  - Device handle
- Output:
  - Device statistics
  - Number of received packets
  - Number of received bytes
  - Number of transmitted packets
  - Number of transmitted bytes
  - Number of receive errors
  - Number of transmit errors
  - Number of multicast packets received

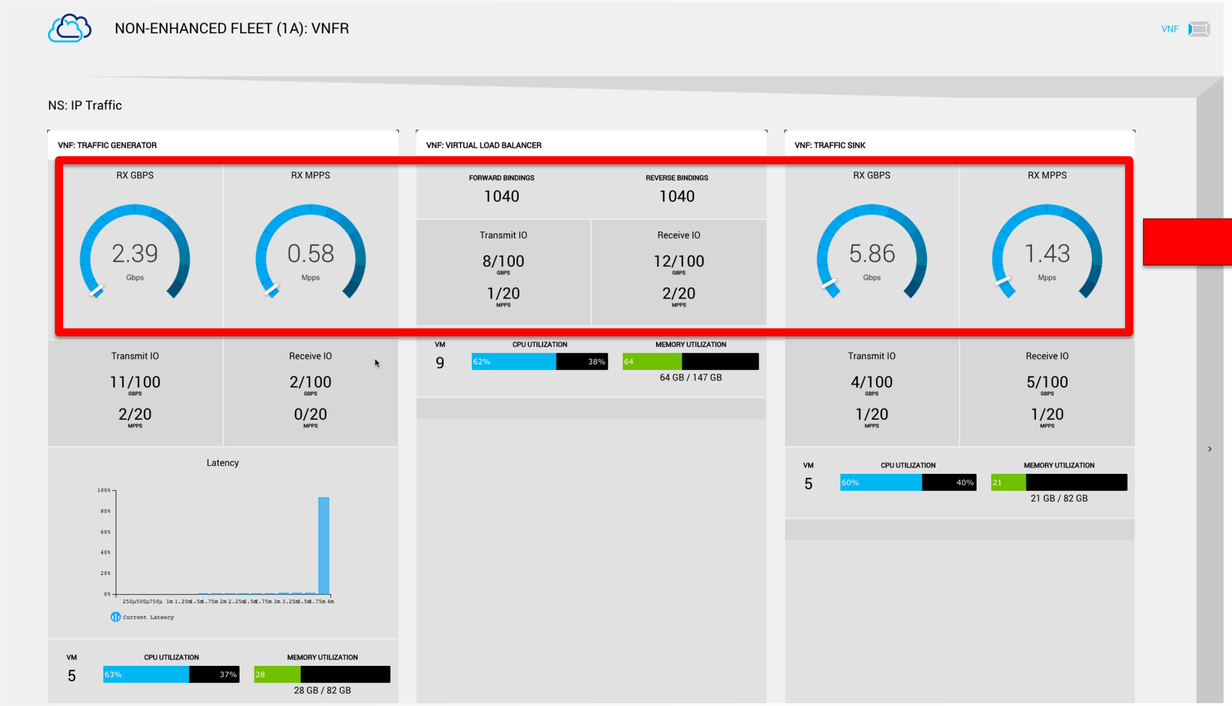
Device Tx Queue Setup - This API call is used to set up the specified transmit queue of the device.

- Input parameters:
  - Device handle:
  - Queue ID
  - Number of Tx descriptors
  - Memory pool for Tx buffer allocation
- Output:
  - Success/failure status

Device Rx Queue Setup - This API call is used to set up the specified receive queue of the device.

- Input parameters:
  - Device handle
  - Queue ID
  - Number of Rx descriptors
  - Memory pool for Rx buffer allocation
- Output:
  - Success/failure status

An Example Use Case of a DPDK Packet I/O Toolkit Performance



Non-DPDK Enabled Network Service (Fleet)

- Traffic generator VNF → Virtual Load Balancer VNF → Traffic Sink/Reflector VNF
- Intel Niantic NICs in Wildcat Pass servers running over Cisco Nexus 3K switches
- Virtio connection to OVS with out DPDK on all hosts

DPDK Enabled Network Service (Fleet)

- Traffic generator VNF → Virtual Load Balancer VNF → Traffic Sink/Reflector VNF
- Intel Niantic NICs in Wildcat Pass servers running over Cisco Nexus 3K switches
- DPDK Enabled OVS using virtio driver on all hosts

Summary Slide

- Virtual Network Functions (VNFs) are diverse and will have many different network connectivity requirements
  - PCI Pass-Through / SR-IOV / KNI / ivshmem / virt-io / vhost-user
- Network connectivity options, in combination with other enhancement choices, have a dramatic effect on performance
  - NUMA, CPU Pinning, Huge Pages, CAT/CMT, QAT
- Leveraging DPDK to build a network abstraction layer (Packet I/O Toolkit) provides simplified VNF networking
  - Write once, deploy anywhere

Thank You