**1 Introduction**

This guide is a Reference Architecture document that provides instructions on how to build a Bare Metal Container setup on an Intel platform using accelerators. The goal of this guide is to help developers adopt and use advanced networking technologies and device plugin features in a container bare metal reference architecture. This guide includes coverage for the 2nd generation Intel® Xeon® Scalable processors (formerly codenamed Cascade Lake). The container reference architecture represents a baseline configuration of components that are combined to achieve optimal system performance for applications running in a container-based environment.

This document contains a description of configuration, installation, and use of networking and device plugin features for Kubernetes. The reference architecture includes a playbook, which enables users to perform automated deployments, thus decreasing installation time from days to hours.

This document is part of the Network Transformation Experience Kit, which is available at: [https://networkbuilders.intel.com/](https://networkbuilders.intel.com/)

Containers are one of the hottest technologies in cloud computing and fundamental to Cloud Native adoption. More so than the virtual machine, a container is lightweight, agile and portable. It can be quickly created, updated, and removed. Kubernetes* (K8s*) is the leading open source system for automating deployment, scaling, and management of containerized applications. To enhance Kubernetes for network functions virtualization (NFV) and networking usage, Intel and its partners are developing the following:

- Suite of capabilities and methodologies that exposes Intel® Architecture platform features for increased and deterministic application and network performance.
- Networking features in Kubernetes including Multus, a plugin that provides multiple network interfaces within a Kubernetes container, SR-IOV CNI plugin, and Userspace CNI plugin to improve data throughput.
- Device plugin features in Kubernetes including GPU, FPGA, Intel® QuickAssist Technology, and SR-IOV device plugins to boost performance and platform efficiency.
Table of Contents

1 Introduction ................................................................................................................................................. 1
1.1 Technology overview ............................................................................................................................ 3
1.2 Terminology ............................................................................................................................................ 3
1.3 Reference documents ............................................................................................................................ 4

2 Physical and Software topology .............................................................................................................. 4
2.1 Physical topology .................................................................................................................................. 4
2.2 Container Bare Metal Reference Architecture (BMRA) Ansible* Playbook ..................................... 5

3 Hardware BOM ........................................................................................................................................... 7

4 Software BOM ............................................................................................................................................ 8
4.1 Platform BIOS settings ........................................................................................................................... 9

5 System prerequisites ............................................................................................................................ 9
5.1 Master and minion BIOS prerequisites .............................................................................................. 9
5.2 Master and minion nodes network interface requirements ............................................................ 10
5.3 Ansible Host, master, and minion software prerequisites .............................................................. 10

6 Deploy Intel Bare Metal Reference Architecture using Ansible Playbook ........................................... 10
6.1 Get Ansible Playbook and modify variables .................................................................................. 10
6.2 Execute the Ansible playbook ........................................................................................................... 13

7 Post-deployment verification .................................................................................................................. 13
7.1 Check the Kubernetes cluster .......................................................................................................... 13
7.2 Check Node Feature Discovery (NFD) .............................................................................................. 15
7.3 Check CPU Manager for Kubernetes ............................................................................................... 17
7.4 Intel Device Plugins for Kubernetes ............................................................................................... 18
7.4.1 SR-IOV network device plugin ................................................................................................... 18
7.4.2 Check QAT device plugin ........................................................................................................... 19
7.5 Verify Advanced Networking Features (after Installation) ............................................................ 19
7.5.1 Multus CNI plugin ....................................................................................................................... 19
7.5.2 SR-IOV CNI plugin ....................................................................................................................... 19
7.5.3 Userspace CNI plugin ................................................................................................................... 20

8 Use cases .................................................................................................................................................. 20
8.1 Use NFD to select the node to deploy the pod .................................................................................. 20
8.2 Pod using SR-IOV ................................................................................................................................ 20
8.2.1 Pod using SR-IOV DPDK ........................................................................................................... 20
8.2.2 Pod using SR-IOV CNI plugin ..................................................................................................... 21
8.3 Pod using Userspace CNI plugin ........................................................................................................ 23
8.3.1 Pod using Userspace CNI with OVS-DPDK ............................................................................... 23
8.4 Pod using CPU Pinning and Isolation in Kubernetes ........................................................................ 27

9 Conclusion – Automation Eases Reference Application Deployment .................................................. 28

Figures

Figure 1. Reference Architecture Topology.................................................................................................. 5
Figure 2. BMRA High Level Ansible Playbook ......................................................................................... 6
Figure 3. BMRA Detailed Ansible Playbook ............................................................................................. 6
Figure 4. Features detected by NFD ......................................................................................................... 16

Tables

Table 1. Terminology .................................................................................................................................. 3
Table 2. Reference documents .................................................................................................................. 4
Table 3. Hardware BOM ............................................................................................................................ 7
Table 4. Software BOM ............................................................................................................................... 8
Table 5. Platform BIOS settings .............................................................................................................. 9
Table 6. Group Variables ........................................................................................................................... 11
Table 7. Host Variables ............................................................................................................................. 12
1.1 Technology overview

The technology described in this document consists of a suite of capabilities implemented across the NFVI stack, which enables a Kubernetes environment that leverages Intel technologies and targets intelligent platform capability, configuration, and capacity data consumption. Intel and partners have worked together to progress the discovery, scheduling, and isolation of server hardware features using the following technologies:

- Node Feature Discovery (NFD) enables generic hardware capability discovery in Kubernetes, including Intel® Xeon® processor-based hardware.
- CPU Manager for Kubernetes provides a mechanism for CPU core pinning and isolation of containerized workloads.
- Huge page support, added to Kubernetes v1.8, enables the discovery, scheduling and allocation of huge pages as a native first-class resource. This support addresses low latency and deterministic memory access requirements.
- SR-IOV provides I/O virtualization that makes a single PCIe device (typically a NIC) appear as many network devices in the Linux* kernel. In Kubernetes this results in network connections that can be separately managed and assigned to different pods.
- Kubernetes Device Plugin Framework and device plugins, including GPU, FPGA, Intel® QuickAssist Technology, and SR-IOV device plugins to boost performance and platform efficiency.

One of the important parts in the Advanced Networking Features is Multus, which supports multiple network interfaces per pod to expand the networking capability of Kubernetes. Supporting multiple network interfaces is a key requirement for many virtual network functions (VNFs), as they require separation of control, management, and data planes. Multiple network interfaces are also used to support different protocols or software stacks and different tuning and configuration requirements.

Advanced Networking Features also introduced the SR-IOV CNI plugin and Userspace CNI plugin to enable high performance networking for container-based applications. The SR-IOV CNI plugin allows a Kubernetes pod to be attached directly to a SR-IOV virtual function (VF) using the standard SR-IOV VF driver in the container host's kernel. The Userspace CNI plugin was designed to implement userspace networking (as opposed to kernel space networking), like DPDK based applications. It can run with vSwitches such as OVS-DPDK or VPP and provides a high performance container networking solution through dataplane acceleration in NFV environments.

With the Kubernetes Device Plugin Framework, Intel provides several device plugins to free up CPU cycles and boost performance. It can deliver efficient acceleration of graphics, compute, data processing, security, and compression. These device plugins include:

- GPU device plugin: VNFs can take advantage of storing, streaming, and transcoding with the Intel GPU device plugin. Intel® Graphics Technology and Intel® Quick Sync Video Technology can accelerate graphics performance.
- FPGA device plugin: Scalable and programmable acceleration in a broad array of applications such as communications, data center, military, broadcast, automotive, and other end markets.
- Intel® QuickAssist Technology (Intel® QAT) device plugin: Directs crypto and data compression functionality to dedicated hardware, accelerating bulk crypto, public key encryption, and compression on Intel® architecture platforms.
- SR-IOV device plugin: supports DPDK VNFs that execute the VF driver and network protocol stack in userspace.

1.2 Terminology

Table 1. Terminology

<table>
<thead>
<tr>
<th>ABBREVIATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIOS</td>
<td>Basic Input / Output System</td>
</tr>
<tr>
<td>BMRA</td>
<td>Bare Metal Reference Architecture</td>
</tr>
<tr>
<td>CNI</td>
<td>Container Networking Interface</td>
</tr>
<tr>
<td>DHCP</td>
<td>Dynamic Host Configuration Protocol</td>
</tr>
<tr>
<td>DPKD</td>
<td>Data Plane Development Kit</td>
</tr>
<tr>
<td>HA</td>
<td>High Availability</td>
</tr>
<tr>
<td>IA</td>
<td>Intel® Architecture</td>
</tr>
<tr>
<td>Intel® HT Technology</td>
<td>Intel® Hyper-Threading Technology</td>
</tr>
<tr>
<td>Intel® QAT</td>
<td>Intel® QuickAssist Technology</td>
</tr>
<tr>
<td>Intel® VT-d</td>
<td>Intel® Virtualization Technology (Intel® VT) for Directed I/O</td>
</tr>
<tr>
<td>Intel® VT-x</td>
<td>Intel® Virtualization Technology (Intel® VT) for IA-32, Intel® 64 and Intel® Architecture</td>
</tr>
<tr>
<td>KBs*</td>
<td>Kubernetes*</td>
</tr>
<tr>
<td>NFD</td>
<td>Node Feature Discovery</td>
</tr>
<tr>
<td>NFV</td>
<td>Network Functions Virtualization</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
</tbody>
</table>
### 1.3 Reference documents

<table>
<thead>
<tr>
<th>REFERENCE</th>
<th>SOURCE</th>
</tr>
</thead>
</table>

### 2 Physical and Software topology

#### 2.1 Physical topology

This section describes the physical topology for the Intel Container Bare Metal Reference Architecture, which uses 3 master nodes and 2 worker nodes in a cluster setup. The diagram also shows the advanced network features and technologies that were used, including CPU Manager for Kubernetes (CMK), Node Feature Discovery (NFD), and SR-IOV device plugin.

This Reference Architecture uses testPMD and pktgen pods and sample VNF workloads (vCMTS and vBNG). Figure 1 shows the physical topology for the reference architecture described in this document.
Intel provides a set of Ansible* and Helm* scripts which enable easy and fast automatic installation on a container bare metal NFV Platform. Ansible is an agentless configuration management tool that uses playbooks to perform actions on many machines and Helm is a package manager tool which runs on top of Kubernetes (K8s) to automate the installation process of plugins and K8s capabilities. The installation is done using the master Ansible playbook which performs initial system configuration, deploys Kubernetes and its add-ons via KubeSpray followed by the K8s capabilities. The playbook enables users to customize multiple parameters to fit their installation requirements.

**Figure 2** shows the three high-level Ansible playbook components, which include:

1. Infrastructure setup block
2. Kubernetes setup block
3. Capabilities setup block
Ansible requires the following components to work: Inventory files, group_vars, host_vars, playbooks, roles and tasks. The user can customize multiple parameters in these components to fit their installation requirements. Refer to Section 6 for details.

**Figure 2. BMRA High Level Ansible Playbook**

Figure 3 describes the detailed Ansible playbook components for software installation described in this document.

**Figure 3. BMRA Detailed Ansible Playbook**

The Ansible playbook runs three sub playbooks, as shown in Figure 3. The 3 sub playbooks in order of their installation are:

1. Infrastructure Set Up (named infra playbook): This playbook modifies kernel boot parameters and sets the initial system configuration in the cluster.
   - Uses isolcpus boot parameter to ensure exclusive cores in CPU Manager for Kubernetes are not affected by other system tasks.
   - Adds the isolcpus in grub file.
   - I/O Memory Management Unit (IOMMU) support is not enabled by default in the CentOS* 7.0 distribution, however, IOMMU support is required for a VF to function properly when assigned to a virtual environment, such as a VM or a container.
   - Enables IOMMU support for Linux kernels.

2. Kubernetes Setup (K8s):

3. Capability Setup:

   - Multus CNI Plugin
   - Other CNI Plugins (SR-IOV & Userspace)
   - CPU Manager for Kubernetes (Helm chart)
   - Node Feature Discovery (Helm chart)
   - SR-IOV Network Device Plugin (Helm chart)
   - Intel Device Plugins for Kubernetes, Intel QAT & GPU (Helm chart)
Reference Architecture | Container Bare Metal for 2nd Generation Intel® Xeon® Scalable Processor

- User can set the size and number of hugepages as per requirement. The default size is set to 2M which can be reconfigurable.
- SR-IOV virtual functions are created by writing an appropriate value to the sriov_numvfs parameter. Appropriate SR-IOV PCI address must be used to fit the environment.

2. Kubernetes Set Up (named k8s playbook): This playbook deploys a High Availability (HA) k8s cluster using Kubespray which is a project under the Kubernetes community that deploys production-ready Kubernetes clusters. The Multus CNI plugin which is specifically designed to provide support for multiple networking interfaces in a k8s environment is deployed by Kubespray along with Flannel, Docker registry and Helm server.

3. Intel BM RA Capability Setup (named Intel playbook): Advanced networking technologies, Enhanced Platform Awareness, and device plugin features are deployed by this playbook using Helm Charts as part of the BM RA. The following capabilities are deployed:
   - Multus CNI plugin. Though Multus is deployed as part of KubeSpray in the K8s playbook, this gives the user the additional benefit to use a different version of Multus to the one supported by KubeSpray and also allows the user the flexibility to install only the Intel capabilities on top of an existing K8s cluster if the K8s playbook is not required.
   - SR-IOV CNI plugin which allows a Kubernetes pod to be attached directly to an SR-IOV virtual function (VF) using the standard SR-IOV VF driver in the container host’s kernel.
   - Userspace CNI plugin which is a Container Network Interface plugin designed to implement user space networking such as DPDK based applications.
   - CPU Manager for K8s (CMK) which performs a variety of operations to enable core pinning and isolation on a container or a thread level.
   - Node Feature Discovery (NFD) which is a k8s add-on to detect and advertise hardware and software capabilities of a platform that can, in turn, be used to facilitate intelligent scheduling of a workload.
   - SR-IOV network device plugin which discovers and exposes SR-IOV network resources as consumable extended resources in Kubernetes.
   - Intel QuickAssist Technology (Intel QAT) device plugin in which a workload can alleviate pipeline bottlenecks and improve performance by offloading cryptographic operations of user traffic to an Intel QAT device.
   - GPU device plugin which offloads the processing of computation intensive workloads to GPU hardware.

3. Hardware BOM

This section lists the hardware components and systems that were utilized in this reference architecture. 2nd Generation Intel® Xeon® Scalable processors feature a scalable, open architecture designed for the convergence of key workloads such as applications and services, control plane processing, high-performance packet processing, and signal processing.

<table>
<thead>
<tr>
<th>ITEM</th>
<th>DESCRIPTION</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Intel® Xeon® Processor Scalable Family</td>
<td>Intel® Xeon® processor-based dual-processor server board</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 x 25 GbE integrated LAN ports</td>
</tr>
<tr>
<td>Processors</td>
<td>6x Intel® Xeon® Gold 5218N Processor</td>
<td>16 cores, 32 threads, 2.3 GHz, 105 W, 38.5 MB L3 total cache per processor, 3 UPI Links, DDR4-2666, 6 memory channels</td>
</tr>
<tr>
<td></td>
<td>2x Intel® Xeon® Gold 6230N Processor</td>
<td>20 cores, 40 threads, 2.0 GHz, 125 W, 27.5 MB L3 cache per processor, 3 UPI Links, DDR4-2666, 6 memory channels</td>
</tr>
<tr>
<td></td>
<td>2x Intel® Xeon® Gold 6252N Processor</td>
<td>24 cores, 48 threads, 2.3 GHz, 150 W, 22 MB L3 cache per processor, 3 UPI Links, DDR4-2666, 6 memory channels</td>
</tr>
<tr>
<td>Memory</td>
<td>192GB (12 x 16GB 2666MHz DDR RDIMM) or minimum all 6 memory channels populated (1 DPC) to achieve 384 GB</td>
<td>192GB to 384GB</td>
</tr>
<tr>
<td>Networking</td>
<td>2 x NICs - Required</td>
<td>2 x Dual Port 25GbE Intel® Ethernet Network Adapter XXV710</td>
</tr>
<tr>
<td></td>
<td>Each NIC NUMA aligned</td>
<td>2 x Dual Port 10GbE Intel® Ethernet Converged Network Adapter X710</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 x Intel® Ethernet Server Adapter X520-DA2 SFP</td>
</tr>
<tr>
<td>Local Storage</td>
<td>2 x &gt;=480GB Intel® SSD SATA or Equivalent Boot Drive. This is for the primary Boot / OS storage. These drives can be sourced by the PCH. These drives should be capable of use in a RAID1 configuration.</td>
<td>2 x Intel® NVMe P4510 Series 2.0TB each Drive recommended NUMA aligned - Required</td>
</tr>
<tr>
<td>Intel® QuickAssist Technology</td>
<td>Intel® C620 Series Chipset Integrated on baseboard Intel® C627/C628 Chipset</td>
<td>Integrated w/NUMA connectivity to each CPU or minimum 16 Peripheral Component Interconnect express* (PCIe*) lane Connectivity to one CPU</td>
</tr>
</tbody>
</table>
## BIOS

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Corporation SE5C620.86 B.0D.01.0241</td>
<td>Intel® Hyper-Threading Technology (Intel® HT Technology) enabled</td>
</tr>
<tr>
<td>Release Date: 11/19/2018</td>
<td>Intel® Virtualization Technology (Intel® VT-x) enabled</td>
</tr>
<tr>
<td></td>
<td>Intel® Virtualization Technology for Directed I/O (Intel® VT-d) enabled</td>
</tr>
</tbody>
</table>

## Switches

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cisco® Catalyst 2960-XR</td>
<td>Cisco 1GbE Switch</td>
</tr>
<tr>
<td>Arista® DCS-7280QR-C36-R</td>
<td>Arista 25GbE Switch</td>
</tr>
</tbody>
</table>

## 4 Software BOM

### Table 4. Software BOM

<table>
<thead>
<tr>
<th>SOFTWARE FUNCTION</th>
<th>SOFTWARE COMPONENT</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host OS</td>
<td>CentOS® 7.6 build 1810 Kernel version: 3.10.0-957.1.3.el7.x86_64</td>
<td><a href="https://www.centos.org/">https://www.centos.org/</a></td>
</tr>
<tr>
<td>Ansible*</td>
<td>Ansible v2.7.1</td>
<td><a href="https://www.ansible.com/">https://www.ansible.com/</a></td>
</tr>
<tr>
<td>BMRA Ansible Playbook</td>
<td>Master Playbook v1.0</td>
<td><a href="https://github.com/intel/container-experience-kits">https://github.com/intel/container-experience-kits</a></td>
</tr>
<tr>
<td>Python*</td>
<td>Python 2.7</td>
<td><a href="https://www.python.org/">https://www.python.org/</a></td>
</tr>
<tr>
<td>Kubespray*</td>
<td>Kubespray: v2.8.0-31-g3c44fffc</td>
<td><a href="https://github.com/kubernetes-sigs/kubespray">https://github.com/kubernetes-sigs/kubespray</a></td>
</tr>
<tr>
<td>Docker*</td>
<td>Docker* 18.6.2</td>
<td><a href="https://www.docker.com/">https://www.docker.com/</a></td>
</tr>
<tr>
<td>Container orchestration engine</td>
<td>Kubernetes* v1.13.5</td>
<td><a href="https://github.com/kubernetes/kubernetes">https://github.com/kubernetes/kubernetes</a></td>
</tr>
<tr>
<td>CPU Manager for Kubernetes*</td>
<td>CPU Manager for Kubernetes* v1.3.1</td>
<td><a href="https://github.com/intel/CPU-Manager-for-Kubernetes">https://github.com/intel/CPU-Manager-for-Kubernetes</a></td>
</tr>
<tr>
<td>Node Feature Discovery</td>
<td>NFD v0.4.0</td>
<td><a href="https://github.com/kubernetes-sigs/node-feature-discovery">https://github.com/kubernetes-sigs/node-feature-discovery</a></td>
</tr>
<tr>
<td>Data Plane Development Kit</td>
<td>DPDK 18.11.1</td>
<td><a href="http://dpdk.org/git/dpdk">http://dpdk.org/git/dpdk</a></td>
</tr>
<tr>
<td>Open vSwitch with DPDK</td>
<td>OVS-DPDK 2.11.90</td>
<td><a href="http://docs.openvswitch.org/en/latest/intro/install/dpdk/">http://docs.openvswitch.org/en/latest/intro/install/dpdk/</a></td>
</tr>
<tr>
<td>Vector Packet Processing</td>
<td>VPP 19.01</td>
<td><a href="https://docs.fd.io/vpp/19.01/index.html">https://docs.fd.io/vpp/19.01/index.html</a></td>
</tr>
<tr>
<td>Multus CNI</td>
<td>Multus CNI v3.2</td>
<td><a href="https://github.com/intel/multus-cni">https://github.com/intel/multus-cni</a></td>
</tr>
<tr>
<td>SR-IOV CNI</td>
<td>SR-IOV CNI v2.0.0</td>
<td><a href="https://github.com/intel/SR-IOV-network-device-plugin">https://github.com/intel/SR-IOV-network-device-plugin</a></td>
</tr>
<tr>
<td>Userspace CNI</td>
<td>Userspace CNI v1.1</td>
<td><a href="https://github.com/intel/userspace-cni-network-plugin">https://github.com/intel/userspace-cni-network-plugin</a></td>
</tr>
</tbody>
</table>
| Intel Ethernet Drivers | | https://sourceforge.net/projects/e1000/files/ixgbe%20stable/5.2.1  
https://sourceforge.net/projects/e1000/files/ixgbevf%20stable/4.2.1  
https://sourceforge.net/projects/e1000/files/i40e%20stable/2.0.30  
https://sourceforge.net/projects/e1000/files/i40evf%20stable/2.0.30 |
4.1 Platform BIOS settings

Table 5. Platform BIOS settings

<table>
<thead>
<tr>
<th>MENU (ADVANCED)</th>
<th>PATH TO BIOS SETTING</th>
<th>BIOS SETTING</th>
<th>SETTINGS FOR DETERMINISTIC PERFORMANCE</th>
<th>SETTINGS FOR MAX PERFORMANCE WITH TURBO MODE ENABLED</th>
<th>REQUIRED OR RECOMMENDED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Configuration</td>
<td>CPU P State Control</td>
<td>EIST PSD Function</td>
<td>HW_ALL</td>
<td>SW_ALL</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boot Performance Mode</td>
<td>Max. Performance</td>
<td>Max. Performance</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Energy Efficient Turbo</td>
<td>Disable</td>
<td>Disable</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Turbo Mode</td>
<td>Disable</td>
<td>Enable</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel® SpeedStep® (Pstates) Technology</td>
<td>Disable</td>
<td>Enable</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>Hardware PM State Control</td>
<td>Hardware P-States</td>
<td>Disable</td>
<td>Disable</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>CPU C State Control</td>
<td>Autonomous Core C-State</td>
<td>Disable</td>
<td>Enable</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU C6 Report</td>
<td>Disable</td>
<td>Disable</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enhanced Halt State (C1E)</td>
<td>Disable</td>
<td>Enable</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>Energy Perf Bias</td>
<td>Power Performance Tuning</td>
<td>BIOS Controls EPB</td>
<td>BIOS Controls EPB</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENERGY_PERF_BIAS_CFG Mode</td>
<td>Perf</td>
<td>Perf</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>Package C State Control</td>
<td>Package C State</td>
<td>CO/C1 State</td>
<td>C6</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>Intel® UPI General Configuration</td>
<td>LINK LOP ENABLE</td>
<td>Disable</td>
<td>Disable</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LINK L1 ENABLE</td>
<td>Disable</td>
<td>Disable</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SNC</td>
<td>Disable</td>
<td>Disable</td>
<td>Required</td>
</tr>
<tr>
<td>Memory Configuration</td>
<td>Enforce POR</td>
<td>Disable</td>
<td>Disable</td>
<td>Disble</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>IMC Interleaving</td>
<td>2-Way Interleave</td>
<td>2-Way Interleave</td>
<td>Recommended</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volatile Memory Mode</td>
<td>2 LM mode</td>
<td>2 LM mode</td>
<td>Required</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Force 1-Ch Way in FM</td>
<td>Disabled</td>
<td>Disabled</td>
<td>Required</td>
<td></td>
</tr>
<tr>
<td>Platform Configuration</td>
<td>Miscellaneous Configuration</td>
<td>Serial Debug Message Level</td>
<td>Minimum</td>
<td>Minimum</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>PCI Express*</td>
<td>PCIe* ASPM Support</td>
<td>Per Port</td>
<td>Per Port</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>Uncore</td>
<td>Uncore Frequency Scaling</td>
<td>Disable</td>
<td>Disable</td>
<td>Required</td>
</tr>
</tbody>
</table>

Note: To gather performance data required for conformance, use either column with deterministic performance or turbo mode enabled in this table. Some solutions may not provide the BIOS options that are documented in this table. For Intel® Select Solution, the BIOS should be set to the “Max Performance” profile with Virtualization.

5 System prerequisites

This section describes the minimal system prerequisites needed for the Ansible Host and master/minion nodes.

5.1 Master and minion BIOS prerequisites

Enter the BIOS menu and update the configuration as follows:

<table>
<thead>
<tr>
<th>PACKAGE</th>
<th>ANSIBLE HOST</th>
<th>MASTER/MINION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® VT-x</td>
<td>Enabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>Intel® HT Technology</td>
<td>Enabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>Intel® VT-d (SR-IOV)</td>
<td>Enabled</td>
<td>Enabled</td>
</tr>
</tbody>
</table>
5.2 Master and minion nodes network interface requirements

The following list provides a brief description of different networks and network interfaces used in the lab setup.

- Internet network
  - Ansible Host accessible
  - Capable of downloading packages from the internet
  - Can be configured for Dynamic Host Configuration Protocol (DHCP) or with static IP address
- Management network and flannel pod network interface
  - Kubernetes master and minion nodes inter-node communications
  - Flannel pod network will run over this network
  - Configured to use a private static address
- Tenant data network(s)
  - Dedicated networks for traffic
  - SR-IOV enabled
  - VF can be DPDK bound in pod

5.3 Ansible Host, master, and minion software prerequisites

1. Enter the following commands in Ansible Host:
   ```
   # sudo yum install epel-release
   # sudo yum install ansible
   # easy_install pip
   # pip2 install jinja2 --upgrade
   # sudo yum install python36 -y
   ```

2. Enable passwordless login between all nodes in the cluster.
   Step 1: Create Authentication SSH-Keypair Keys on Ansible Host:
   ```
   # ssh-keygen
   ```
   Step 2: Upload Generated Public Keys to all the nodes from Ansible Host:
   ```
   # ssh-copy-id root@node-ip-address
   ```

6 Deploy Intel Bare Metal Reference Architecture using Ansible Playbook

6.1 Get Ansible Playbook and modify variables

1. Get Ansible playbook:
   ```
   # git clone https://github.com/intel/container-experience-kits.git
   # cd container-experience-kits/playbooks
   ```

2. Copy example inventory file to the playbook home location:
   ```
   # cp examples/inventory.ini .
   ```

3. Edit the inventory.ini to reflect the requirement. Here is the sample file.
   ```
   [all]
   node1 ansible_host=10.250.250.161 ip=10.250.250.161
   node2 ansible_host=10.250.250.162 ip=10.250.250.162
   node3 ansible_host=10.250.250.163 ip=10.250.250.163
   node4 ansible_host=10.250.250.166 ip=10.250.250.166
   node5 ansible_host=10.250.250.167 ip=10.250.250.167
   [kube-master]
   node1
   node2
   node3
   [etcd]
   node1
   node2
   node3
   [kube-node]
   node5
   [k8s-cluster:children]
   kube-master
   kube-node
   [calico-rr]
   ```

4. Copy group_vars and host_vars directories to the playbook home location:
   ```
   # cp -r examples/group_vars examples/host_vars .
   ```

5. Update group_vars to match the desired configuration. Use Table 6 as a reference.
# vim group_vars/all.yml

---

## BMRA master playbook variables ##

# Node Feature Discovery
nfd_enabled: true
nfd_build_image_locally: true
nfd_namespace: kube-system
nfd_sleep_interval: 30s

# Intel CPU Manager for Kubernetes
cmk_enabled: true
cmk_namespace: kube-system

#cmk_hosts_list: node1,node2 # allows to control where CMK nodes will run, leave this option commented out to deploy on all K8s nodes

# Intel SRIOV Network Device Plugin
sriov_net_dp_enabled: true
sriov_net_dp_namespace: kube-system

# whether to build and store image locally or use one from public external registry
sriov_net_dp_build_image_locally: true

# Intel Device Plugins for Kubernetes
qat_dp_enabled: true
qat_dp_namespace: kube-system
gpu_dp_enabled: true
gpu_dp_namespace: kube-system

# Forces installation of the Multus CNI from the official Github repo on top of the Kubespray built-in one
force_external_multus_installation: true

## Proxy configuration ##

## Kubespray variables ##

# default network plugins and kube-proxy configuration
kube_network_plugin_multus: true
multus_version: v3.2

*Note:* Please pay special attention to the ‘http_proxy’, ‘https_proxy’ and ‘no_proxy’ variables if operated from behind proxy. Add all cluster node IP addresses to ‘no_proxy’ variable.

### Table 6. Group Variables

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>CHOICES/DEFAULTS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>nfd_enabled</td>
<td>Boolean, default “true”</td>
<td>Install NFD on the target cluster</td>
</tr>
<tr>
<td>nfd_build_image_locally</td>
<td>Boolean, default “true”</td>
<td>Build and host NFD image in the cluster local Docker registry, set to false to use NFD image from quay.io</td>
</tr>
<tr>
<td>nfd_namespace</td>
<td>String, default “kube-system”</td>
<td>Kubernetes namespace to be used for NFD deployment</td>
</tr>
<tr>
<td>cmk_enabled</td>
<td>Boolean, default “true”</td>
<td>Install CMK on the target cluster</td>
</tr>
<tr>
<td>cmk_namespace</td>
<td>String, default “kube-system”</td>
<td>Kubernetes namespace to be used for CMK deployment</td>
</tr>
<tr>
<td>cmk_use_all_hosts</td>
<td>Boolean, default “true”</td>
<td>Use --all-hosts argument for CMK cluster-init, effectively deploying CMK on all Kubernetes nodes, if set to “false”, cmk_hosts_list must be set</td>
</tr>
</tbody>
</table>
### Table 7. Host Variables

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>CHOICES/Defaults</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmk_hosts_list</td>
<td>Comma separated list of node names</td>
<td>List of node names to deploy CMK on, ignored if cmk_use_all_hosts set to true</td>
</tr>
<tr>
<td>cmk_shared_num_cores</td>
<td>Integer, default 2</td>
<td>Number of CPU cores to be added to the shared pool</td>
</tr>
<tr>
<td>cmk-exclusive_num_cores</td>
<td>Integer, default: 2</td>
<td>Number of CPU cores to be added to the exclusive pool</td>
</tr>
<tr>
<td>cmk_shared_mode</td>
<td>“packed” or “shared”, default: “packed”</td>
<td>Allocation mode for shared pool (how cores are allocated across NUMA nodes)</td>
</tr>
<tr>
<td>cmk-exclusive_mode</td>
<td>“packed” or “shared”, default: “packed”</td>
<td>Allocation mode for exclusive pool (how cores are allocated across NUMA nodes)</td>
</tr>
<tr>
<td>sriov_net_dp_enabled</td>
<td>Boolean, default “true”</td>
<td>Install SROIV Network Device Plugin on the target cluster</td>
</tr>
<tr>
<td>sriov_net_dp_namespace</td>
<td>String, default “kube-system”</td>
<td>Kubernetes namespace to be used for SROIV Network Device Plugin deployment</td>
</tr>
<tr>
<td>sriov_net_dp_build_image_locally</td>
<td>Boolean, default “true”</td>
<td>Build and host SROIV Network Device Plugin image in the cluster local Docker registry, set to false to use SROIV Network Device Plugin image from Docker Hub</td>
</tr>
<tr>
<td>qat_dp_enabled</td>
<td>Boolean, default “true”</td>
<td>Install Intel QAT Device Plugin on the target cluster</td>
</tr>
<tr>
<td>qat_dp_namespace</td>
<td>String, default “kube-system”</td>
<td>Kubernetes namespace to be used for Intel QAT Device Plugin deployment</td>
</tr>
<tr>
<td>gpu_dp_enabled</td>
<td>Boolean, default “true”</td>
<td>Install Intel GPU Device Plugin on the target cluster</td>
</tr>
<tr>
<td>gpu_dp_namespace</td>
<td>String, default “kube-system”</td>
<td>Kubernetes namespace to be used for Intel GPU Device Plugin deployment</td>
</tr>
<tr>
<td>sriov_numvfs</td>
<td>Integer</td>
<td>Number of VFs to be created per PF</td>
</tr>
<tr>
<td>sriov_enabled</td>
<td>Boolean</td>
<td>Enables SROIV VF for interfaces specified in “sriov_nics” array</td>
</tr>
<tr>
<td>sriov_nics</td>
<td>Array of strings, e.g. [“eth1”, “eth2”]</td>
<td>Network interfaces names of the SROIV PFs</td>
</tr>
</tbody>
</table>

6. Update files in the host_vars directory to match the desired configuration. Use Table 7 as a reference.

**Note:** The host_vars folder should contain individual .yml files for all the nodes defined in the inventory file. In the examples we have defined 2 nodes, thus host_vars should contain the respective .yml files for the nodes, for example, node1.yml and node2.yml.

**Note:** The Isolate CPUs from kernel scheduler variable 'isolcpus' should be assigned a number that represents the number of CPUs of the cluster node that we want to allocate.
### Reference Architecture | Container Bare Metal for 2nd Generation Intel® Xeon® Scalable Processor

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>CHOICES/DEFAULTS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>sriov_cni_enabled</td>
<td>Boolean</td>
<td>Whether to install SRIOV CNI plugin on target node</td>
</tr>
<tr>
<td>sriov_net_dp_autogenerate</td>
<td>Boolean</td>
<td>If “true”, config with a single resource pool and all SRIOV NICs from “sriov_nics” will be created, if “false” configuration from “sriov_net_dp_config” will be used</td>
</tr>
<tr>
<td>sriov_net_dp_device_type</td>
<td>String, “netdevice”, “vfio” or “uio”</td>
<td>SRIOV device driver type</td>
</tr>
<tr>
<td>sriov_net_dp_sriov_mode</td>
<td>Boolean. default “false”</td>
<td>Whether the NICs support SR-IOV, “false” for VM deployments, “true” for bare metal installation</td>
</tr>
<tr>
<td>sriov_net_dp_config:</td>
<td>Array of dicts, please see description</td>
<td>Configure as described here: <a href="https://github.com/intel/sriov-network-device-plugin#config-parameters">https://github.com/intel/sriov-network-device-plugin#config-parameters</a> only if “sriov_net_dp_autogenerate” set to “false”</td>
</tr>
<tr>
<td>userspace_cni_enabled</td>
<td>Boolean, default “true”</td>
<td>Enables installation of Userspace CNI plugin on target machine</td>
</tr>
<tr>
<td>vpp_enabled</td>
<td>Boolean, default “true”</td>
<td>Enables installation of virtual userspace network switch, selecting one is recommended</td>
</tr>
<tr>
<td>ovs_dpdk_enabled</td>
<td>Boolean, default “true”</td>
<td>Enables installation of virtual userspace network switch, selecting one is recommended</td>
</tr>
<tr>
<td>force_nic_drivers_update</td>
<td>Boolean, default “true”</td>
<td>Update i40e and i40evf drivers on the target machine</td>
</tr>
<tr>
<td>hugepages_enabled</td>
<td>Boolean, default “true”</td>
<td>Enable huge pages support</td>
</tr>
<tr>
<td>default_hugepage_size</td>
<td>“2M” or “1G”</td>
<td>Default huge page size</td>
</tr>
<tr>
<td>hugepages_1G</td>
<td>Integer</td>
<td>Sets how many huge pages of each size should be created</td>
</tr>
<tr>
<td>hugepages_2M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isolcpus_enabled</td>
<td>Boolean, default “true”</td>
<td>Isolates CPUs from Linux scheduler</td>
</tr>
<tr>
<td>isolcpus</td>
<td>Comma separated list of integers, maybe mixed with ranges, e.g. “1,2,3-6,8-15,24”</td>
<td>List of CPUs to be isolated</td>
</tr>
</tbody>
</table>

**Note:** Not all variables are defined in the group_vars and host_vars directories - more fine-grained control is possible and can be achieved by overriding vars defined on the role level. Inspect files located in “roles/*/vars” and “roles/*/defaults” for more advanced configuration options, but remember - some of these variables are not exposed for a reason!

7. Update and initialize git submodule.

    # git submodule update --init

This git repository has nested submodules to support kubespray installation. The submodule update command will recurse into registered submodules.

6.2 Execute the Ansible playbook

Deploy Intel Bare Metal Reference Architecture with the command:

    #ansible-playbook -i inventory.ini playbooks/cluster.yml

7 Post-deployment verification

This section shows how to verify all the components deployed by the scripts.

7.1 Check the Kubernetes cluster

1. Check the post-deployment node status of master & minion:

    # kubectl get nodes -o wide

    | NAME     | STATUS   | ROLES   | AGE      | VERSION | INTERNAL-IP | EXTERNAL-IP | OS-IMAGE      |
    |----------|----------|---------|----------|---------|-------------|-------------|---------------|
    | node1    | Ready    | master  | 4d12h    | v1.13.5 | 10.250.250.161 | <none>      | CentOS Linux 7 |
    | (Core)   | 3.10.0-957.12.2.el7.x86_64 | docker://18.6.2 |
    | node2    | Ready    | master  | 4d12h    | v1.13.5 | 10.250.250.162 | <none>      | CentOS Linux 7 |
    | (Core)   | 3.10.0-957.12.2.el7.x86_64 | docker://18.6.2 |
    | node3    | Ready    | master  | 4d12h    | v1.13.5 | 10.250.250.163 | <none>      | CentOS Linux 7 |
    | (Core)   | 3.10.0-957.12.2.el7.x86_64 | docker://18.6.2 |
    | node4    | Ready    | master  | 4d12h    | v1.13.5 | 10.250.250.164 | <none>      | CentOS Linux 7 |
    | (Core)   | 3.10.0-957.12.2.el7.x86_64 | docker://18.6.2 |
    | node5    | Ready    | node    | 4d12h    | v1.13.5 | 10.250.250.165 | <none>      | CentOS Linux 7 |
    | (Core)   | 3.10.0-957.12.2.el7.x86_64 | docker://18.6.2 |

    ...
2. **Check pod status of master & minion. All Pod states should be in Running or Completed status.**

```bash
# kubectl get pods --all-namespaces
```

<table>
<thead>
<tr>
<th>NAMESPACE</th>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>kube-system</td>
<td>cmk-cmk-cluster-init-pod</td>
<td>0/1</td>
<td>Completed</td>
</tr>
<tr>
<td></td>
<td>cmk-init-install-discover-pod-node4</td>
<td>0/2</td>
<td>Completed</td>
</tr>
<tr>
<td></td>
<td>cmk-init-install-discover-pod-node5</td>
<td>0/2</td>
<td>Completed</td>
</tr>
<tr>
<td></td>
<td>cmk-init-install-discover-pod-node6</td>
<td>0/2</td>
<td>Completed</td>
</tr>
<tr>
<td></td>
<td>cmk-reconcile-nodereport-ds-node4-s7dzx</td>
<td>2/2</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>cmk-reconcile-nodereport-ds-node5-6d926</td>
<td>2/2</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>cmk-reconcile-nodereport-ds-node6-44sxx</td>
<td>2/2</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>cmk-webhook-deployment-8456cc6f5-kjjfd</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>coredns-644c686c9-b6b8s</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>coredns-644c686c9-nf5c4</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>dns-autoscaler-586f58b8bf-djf4q</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>intel-gpu-plugin-intel-gpu-plugin-5btnx</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>intel-gpu-plugin-intel-gpu-plugin-dbh95</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>intel-gpu-plugin-intel-gpu-plugin-psr9m</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>intel-qat-plugin-intel-qat-plugin-6dgn8</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>intel-qat-plugin-intel-qat-plugin-cmvxg</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>intel-qat-plugin-intel-qat-plugin-qlgx9</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-apiserver-node1</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-apiserver-node2</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-apiserver-node3</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-controller-manager-node1</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-controller-manager-node2</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-controller-manager-node3</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-flannel-ds-amd64-55bbn</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-flannel-ds-amd64-5bhrs</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-flannel-ds-amd64-cwrdh</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-flannel-ds-amd64-kcfz</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-flannel-ds-amd64-mfzwhf</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-multus-ds-amd64-68zhz</td>
<td>1/1</td>
<td>Running</td>
</tr>
<tr>
<td></td>
<td>kube-multus-ds-amd64-9c989</td>
<td>1/1</td>
<td>Running</td>
</tr>
</tbody>
</table>
Node Feature Discovery (NFD) is a Kubernetes add-on that detects and advertises hardware and software capabilities of a platform that can, in turn, be used to facilitate intelligent scheduling of a workload. Node Feature Discovery is one of the Intel technologies that supports targeting of intelligent configuration and capacity consumption of platform capabilities. NFD runs as a separate container on each individual node of the cluster, discovers capabilities of the node, and finally, publishes these as node labels using the Kubernetes API. NFD only handles non-allocatable features.

NFD currently detects the features shown below.
Figure 4. Features detected by NFD

To verify that NFD is running as expected, use the following command:

```bash
# kubectl get ds --all-namespaces | grep nfd-node-feature-discovery
```

To check the labels created by NFD:

```bash
# kubectl label node --list --all
```

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADX</td>
<td>Multi-Precision Add-Carry Instruction Extensions (ADX)</td>
</tr>
<tr>
<td>AESN</td>
<td>Advanced Encryption Standard (AES) New Instructions (AES-NI)</td>
</tr>
<tr>
<td>AVX</td>
<td>Advanced Vector Extensions (AVX)</td>
</tr>
<tr>
<td>AVX2</td>
<td>Advanced Vector Extensions 2 (AVX2)</td>
</tr>
<tr>
<td>BM1</td>
<td>Bit Manipulation Instruction Set 1 (BMI)</td>
</tr>
<tr>
<td>BM2</td>
<td>Bit Manipulation Instruction Set 2 (BMI2)</td>
</tr>
<tr>
<td>SSF4.1</td>
<td>Streaming SIMD Extensions 4.1 (SSE4.1)</td>
</tr>
<tr>
<td>SSF4.2</td>
<td>Streaming SIMD Extensions 4.2 (SSE4.2)</td>
</tr>
<tr>
<td>RDTM0N</td>
<td>Intel RDT Monitoring Technology</td>
</tr>
<tr>
<td>RDTM1M</td>
<td>Intel Cache Monitoring (CMT)</td>
</tr>
<tr>
<td>RDTM0M</td>
<td>Intel Memory Bandwidth Monitoring (MBM)</td>
</tr>
<tr>
<td>RDTL3CA</td>
<td>Intel L3 Cache Allocation Technology</td>
</tr>
<tr>
<td>RDTL2CA</td>
<td>Intel L2 Cache Allocation Technology</td>
</tr>
<tr>
<td>RDTMBCA</td>
<td>Intel Memory Bandwidth Allocation (MBA) Technology</td>
</tr>
<tr>
<td>selinux</td>
<td>selinux is enabled on the node</td>
</tr>
<tr>
<td>nonrotationdisk</td>
<td>Non-rotational disk, like SSD, is present in the node</td>
</tr>
<tr>
<td>enabled</td>
<td>IOMMU is present and enabled in the kernel</td>
</tr>
<tr>
<td>os_release</td>
<td>ID Operating system identifier</td>
</tr>
<tr>
<td>VERSION_ID</td>
<td>Operating system version identifier (e.g. ‘5.7’)</td>
</tr>
<tr>
<td>VERSION_ID_major</td>
<td>First component of the OS version id (e.g. ‘5’)</td>
</tr>
<tr>
<td>VERSION_ID_minor</td>
<td>Second component of the OS version id (e.g. ‘7’)</td>
</tr>
<tr>
<td>SRIOV</td>
<td>capable Single Root Input/Output Virtualization (SR-IOV) enabled Network Interface Card(s) present</td>
</tr>
<tr>
<td>configured</td>
<td>SR-IOV virtual functions have been configured</td>
</tr>
<tr>
<td>version</td>
<td>full Full kernel version as reported by <code>/proc/sys/kernel/osrelease</code> (e.g. ‘4.3.6-7-g123abcded’)</td>
</tr>
<tr>
<td>major</td>
<td>First component of the kernel version (e.g. ‘4’)</td>
</tr>
<tr>
<td>minor</td>
<td>Second component of the kernel version (e.g. ‘3’)</td>
</tr>
<tr>
<td>revision</td>
<td>Third component of the kernel version (e.g. ‘6’)</td>
</tr>
<tr>
<td>power</td>
<td>ss1_bio_enabled Intel® SST-BF (Intel® Speed Select Technology-Base Frequency) enabled</td>
</tr>
<tr>
<td>numa</td>
<td>Multiple memory nodes i.e. NUMA architecture detected</td>
</tr>
<tr>
<td>nv</td>
<td>present NVMe device(s) are present</td>
</tr>
<tr>
<td>nv</td>
<td>dax NVMe Device(s) are present</td>
</tr>
<tr>
<td>node.alpha.kubernetes-incubator.io/nfd-network-sriov-configured=true</td>
<td></td>
</tr>
<tr>
<td>node.alpha.kubernetes-incubator.io/nfd-cpuid-HTT=true</td>
<td></td>
</tr>
<tr>
<td>node.alpha.kubernetes-incubator.io/nfd-cpuid-SSE4.2=true</td>
<td></td>
</tr>
<tr>
<td>node.alpha.kubernetes-incubator.io/nfd-cpuid-AVX=true</td>
<td></td>
</tr>
</tbody>
</table>
Check CPU Manager for Kubernetes

Kubernetes supports CPU and memory first class resources, while also providing basic support for CPU Pinning and Isolation through the native CPU Manager. To aid commercial adoption, Intel has created CPU Manager for Kubernetes, an open source project that introduces additional CPU optimization capabilities. Without CPU Manager for Kubernetes, the kernel task scheduler will treat all CPUs as available for scheduling process threads and regularly preempts executing process threads to give CPU time to other threads. This non-deterministic behavior makes it unsuitable for latency sensitive workloads.

Using the preconfigured isolcpus boot parameter, CPU Manager for Kubernetes can ensure that a CPU (or set of CPUs) is isolated from the kernel scheduler. Then the latency sensitive workload process thread(s) can be pinned to execute on that isolated CPU set only, providing them exclusive access to that CPU set. While beginning to guarantee the deterministic behavior of priority workloads, isolating CPUs also addresses the need to manage resources, which allows multiple VNFs to coexist on the same physical server. The exclusive pool within CPU Manager for Kubernetes assigns entire physical cores exclusively to the requesting container, meaning no other container will have access to the core.

CPU Manager for Kubernetes performs a variety of operations to enable core pinning and isolation on a container or a thread level. These include:

- Discovering the CPU topology of the machine.
- Advertising the resources available via Kubernetes constructs.
- Placing workloads according to their requests.
- Keeping track of the current CPU allocations of the pods, ensuring that an application will receive the requested resources provided they are available.

CPU Manager for Kubernetes will create three distinct pools: exclusive, shared and infra. The exclusive pool is exclusive meaning only a single task may be allocated to a CPU at a time whereas the shared and infra pools are shared such that multiple processes may be allocated to a CPU. Following is an output for successful CPU Manager for Kubernetes deployment and CPU initialization.

In the example setup, HT is enabled, therefore both physical and associated logical processors are isolated. Four cores are allocated for the CPU Manager for Kubernetes exclusive pool and 1 core for the shared pool. The rest of the CPU cores on the system are for
the infra pool. The correct assignment of CPUs can be seen in the following example. We use node5 minion node as an example.
You can change the node name to reflect your environment.

```
# kubectl logs cmk-init-install-discover-pod-node5 --namespace=kube-system init
INFO:root:Writing config to /etc/cmk.
INFO:root:Requested exclusive cores = 8.
INFO:root:Requested shared cores = 8.
INFO:root:Could not read SST-BF label from the node metadata: 'feature.node.kubernetes.io/cpu-power.sst_bf.enabled'
INFO:root:Isolated logical cores:
16,17,18,19,20,21,22,23,40,41,42,43,44,45,46,47,64,65,66,67,68,69,70,71,88,89,90,91,92,93,94,95
INFO:root:Isolated physical cores: 16,17,18,19,20,21,22,23,40
INFO:root:Adding exclusive pool.
INFO:root:Adding cpu list 16,64 from socket 0 to exclusive pool.
INFO:root:Adding cpu list 17,65 from socket 0 to exclusive pool.
INFO:root:Adding cpu list 18,66 from socket 0 to exclusive pool.
INFO:root:Adding cpu list 19,67 from socket 0 to exclusive pool.
INFO:root:Adding cpu list 20,68 from socket 0 to exclusive pool.
INFO:root:Adding cpu list 21,69 from socket 0 to exclusive pool.
INFO:root:Adding cpu list 22,70 from socket 0 to exclusive pool.
INFO:root:Adding cpu list 23,71 from socket 0 to exclusive pool.
INFO:root:Adding cpu list 40,88,41,89,42,90,43,91,44,92,45,93,46,94,47,95 to shared pool.
INFO:root:Adding infra pool.
INFO:root:Adding cpu list 0,48,1,49,2,50,3,51,4,52,5,53,6,54,7,55,8,56,9,57,10,58,11,59,12,60,13,61,14,62,15,63 to infra pool.
INFO:root:Adding cpu list 24,72,25,73,26,74,27,75,28,76,29,77,30,78,31,79,32,80,33,81,34,82,35,83,36,84,37,85,38,86,39,87 to infra pool.
```

On successful run, the allocatable resource list for the node should be updated with resource discovered by the plugin as shown below. Note that the resource name like "cmk.intel.com/exclusive-cores".

```
# kubectl get node node5 -o json | jq '.status.allocatable'
{
  "cmk.intel.com/exclusive-cores": "8",
  "cpu": "95900m",
  "ephemeral-storage": "48294789041",
  "hugepages-1Gi": "0",
  "hugepages-2Mi": "8Gi",
  "intel.com/sriov_net": "32",
  "memory": "187760556Ki",
  "pods": "110",
  "qat.intel.com/generic": "32"
}
```

CPU Manager for Kubernetes ensures exclusivity, therefore the performance of latency sensitive workloads is not impacted by having a noisy neighbor on the system. CPU Manager for Kubernetes can be used along with the other Intel technology capabilities to achieve the improved network I/O, deterministic compute performance, and server platform sharing benefits offered by Intel® Xeon® Processor-based platforms.

### 7.4 Intel Device Plugins for Kubernetes

Like other vendors, Intel provides many hardware devices that deliver efficient acceleration of graphics, computation, data processing, security, and compression. Those devices optimize hardware for specific tasks, which saves CPU cycles for other workloads and typically results in performance gains. The Kubernetes device plugin framework provides a vendor-independent solution for hardware devices. Intel has developed a set of device plugins that comply with the Kubernetes device plugin framework and allow users to request and consume hardware devices across Kubernetes clusters such as Intel® QuickAssist Technology, GPUs, and FPGAs. The detailed documentation and code are available at:

- **Documentation**: [https://builders.intel.com/docs/networkbuilders/intel-device-plugins-for-kubernetes-appnote.pdf](https://builders.intel.com/docs/networkbuilders/intel-device-plugins-for-kubernetes-appnote.pdf)
- **Code**: [https://github.com/intel/intel-device-plugins-for-kubernetes](https://github.com/intel/intel-device-plugins-for-kubernetes)

#### 7.4.1 SR-IOV network device plugin

The Intel SR-IOV Network device plugin discovers and exposes SR-IOV network resources as consumable extended resources in Kubernetes. It works with SR-IOV VFs with both Kernel drivers and DPDK drivers. When a VF is attached with a kernel driver, then the SR-IOV CNI plugin can be used to configure this VF in the Pod. When using the DPDK driver, a VNF application configures this VF as required.
You need the SR-IOV CNI plugin for configuring VFs with the kernel driver in your pod. The DPDK driver supports VNFs that execute the VF driver and network protocol stack in userspace, allowing the application to achieve packet processing performance that greatly exceeds the ability of the kernel network stack.

By default, when VFs are created for Intel X710 NIC, these VFs are registered with the i40evf kernel module. VF with the Linux network driver uses the kernel network stack for all packet processing. To take advantage of user space packet processing with DPDK, a VF needs to be registered with either igb_uio or vfio-pci kernel module. On successful run, the allocatable resource list for the node should be updated with resource discovered by the plugin as shown below. Note that the resource name is appended with the -resource-prefix, for example, "intel.com/sriov_net_A".

```
# kubectl get node node5 -o json | jq '.status.allocatable'
{
  "cmk.intel.com/exclusive-cores": "8",
  "cpu": "95900m",
  "ephemeral-storage": "48294789041",
  "hugepages-1Gi": "0",
  "hugepages-2Mi": "8Gi",
  "intel.com/sriov_net": "32",
  "memory": "187760556Ki",
  "pods": "110",
  "qat.intel.com/generic": "32"
}
```

### 7.4.2 Check QAT device plugin

Intel® QuickAssist adapters integrate hardware acceleration of compute intensive workloads such as bulk cryptography, public key exchange, and compression on Intel® Architecture Platforms. The Intel® QAT device plugin for Kubernetes supports Intel® QuickAssist adapters and includes an example scenario that uses the Data Plane Development Kit (DPDK) drivers. An additional demo that executes an Intel® QAT accelerated OpenSSL* workload with the Kata Containers runtime is also available. For more information, refer to Intel® QuickAssist adapters: [https://www.intel.com/content/www/us/en/ethernetproducts/gigabit-server-adapters/quickassist-adapter-for-servers.html](https://www.intel.com/content/www/us/en/ethernetproducts/gigabit-server-adapters/quickassist-adapter-for-servers.html).

On successful run, the allocatable resource list for the node should be updated with resource discovered by the plugin as shown below. Note that the resource name uses a format similar to " qat.intel.com/generic ".

```
# kubectl get node node5 -o json | jq '.status.allocatable'
{
  "cmk.intel.com/exclusive-cores": "8",
  "cpu": "95900m",
  "ephemeral-storage": "48294789041",
  "hugepages-1Gi": "0",
  "hugepages-2Mi": "8Gi",
  "intel.com/sriov_net": "32",
  "memory": "187760556Ki",
  "pods": "110",
  "qat.intel.com/generic": "32"
}
```

### 7.5 Verify Advanced Networking Features (after Installation)

#### 7.5.1 Multus CNI plugin

Kubernetes natively supports only a single network interface. Multus is a CNI plugin specifically designed to provide support for multiple networking interfaces in a Kubernetes environment. Operationally, Multus behaves as a broker and arbiter of other CNI plugins, meaning it invokes other CNI plugins (such as Flannel, Calico, SR-IOV, or userspace CNI) to do the actual work of creating the network interfaces. Multus v3.0 that has recently been integrated with Kubevirt, officially recognized as a CNCF project, and officially released with Kubespray v2.8.0. More information and source code can be found at: [https://github.com/Intel-Corp/multus-cni](https://github.com/Intel-Corp/multus-cni)

#### 7.5.2 SR-IOV CNI plugin

Intel introduced the SR-IOV CNI plugin to allow a Kubernetes pod to be attached directly to an SR-IOV virtual function (VF) using the standard SR-IOV VF driver in the container host's kernel. Details on the SR-IOV CNI plugin v1.0.0 can be found at: [https://github.com/intel/sriov-cni](https://github.com/intel/sriov-cni)

Verify the networks using the following command:

```
# kubectl get net-attach-def
NAME   AGE
sriov-net  7d
userspace-ovs  7d
userspace-vpp  7d
```
7.5.3 Userspace CNI plugin

The Userspace CNI is a Container Network Interface plugin designed to implement user space networking such as DPDK based applications. The current implementation supports the DPDK-enhanced open vSwitch (OVS-DPDK) and Vector Packet Processing (VPP) along with the Multus CNI plugin in Kubernetes for the bare metal container deployment model. It enhances the high performance container networking solution and dataplane acceleration for NFV environment.

Verify the networks using the following command:

```
# kubectl get net-attach-def
NAME            AGE
sriov-net       7d
userspace-ovs   7d
userspace-vpp   7d
```

8 Use cases

This section provides several typical use cases to utilize the features and advanced networking capabilities of Kubernetes. The examples shown below are limited to one of two pods for simplicity, but can of course be scaled to a full solution as needed. The first example is a brief showcase of NFD, where the list of available nodes is filtered to only include those with SR-IOV configured.

Three types of network connectivity examples are shown: SR-IOV DPDK, SR-IOV CNI, and Userspace CNI. These showcase different methods for attaching pods to networks, depending on the need for inter- or intra-node connectivity. The last example shows how to use CPU Manager for Kubernetes to provide exclusive CPU core pinning to a pod, which can be useful for deterministic performance and behavior.

8.1 Use NFD to select the node to deploy the pod

With the help of NFD, the Kubernetes scheduler can now use the information contained in node labels to deploy pods according to the requirements identified in the pod specification. Using the nodeSelector option in the pod specification and the matching label on the node, a pod can be deployed on a SR-IOV configured node. If the scheduler does not find such a node in the cluster, then the creation of that pod will fail.

```yaml
... nodeSelector:
    "node.alpha.kubernetes-incubator.io/nfd-network-SR-IOV-configured": "true"
...`

Together with other Intel technologies, including device plugins, NFD facilitates workload optimization through resource-aware scheduling. In particular, NFD can benefit workloads that utilize modern vector data processing instructions, require SR-IOV networking, and have specific kernel requirements.

8.2 Pod using SR-IOV

With the SR-IOV network device plugin and advanced network features for Kubernetes, a pod can take advantage of SR-IOV DPDK or SR-IOV CNI plugin to achieve the best network performance. The following examples are useful when connecting outside of a minion node (inter-node), because the interface is attached to the NIC via SR-IOV, either directly or through the CNI plugin.

8.2.1 Pod using SR-IOV DPDK

1. Create the pod spec file sriov-dpdk-res.yaml.

In the example below, two VFs are requested from the ‘sriov_net_b’ resource that was created previously using the SR-IOV network device plugin. This method assigns VFs to the pod, without providing any interface configuration or IPAM. This makes it ideal for DPDK applications, where the configuration is done directly in the application.

```yaml
# cat sriov-dpdk-res.yaml
apiVersion: v1
kind: Pod
metadata:
  name: sriov-dpdk-pod
  labels:
    env: test
spec:
  tolerations:
  - operator: "Exists"
  containers:
  - name: appcntrl
    image: centos/tools
    command: ['/bin/bash', '-c', 'while true; do sleep 300000; done;']
    resources:
      requests:
```

2. Create the pod from the master node using sriov-dpdk-res.yaml:

```
kubectl create -f sriov-dpdk-res.yaml
pod/sriov-dpdk-pod created
```

3. Inspect the pod environment, you can find the VFs PCI address from the environment variables, shown below using the `env` command. The PCI address of the VFs used by the pod are 0000:18:10.3 and 0000:18:10.5 in this example.

```
kubectl exec -ti sriov-dpdk-pod env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=sriov-dpdk-pod
TERM=xterm
PCIDEVICE_INTEL_COM_SRIOV_NET_B=0000:18:10.3,0000:18:10.5
KUBERNETES_PORT=tcp://10.233.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.233.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.233.0.1
KUBERNETES_SERVICE_HOST=10.233.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
container=docker
HOME=/root
```

At this point, the application running in the pod can decide what to do with the VFs.

### 8.2.2 Pod using SR-IOV CNI plugin

1. Create a sample pod specification `sriov-test.yaml` file.

   In this example, two different networks objects are used, `flannel-networkobj` for access and management, and `sriov-net` for the SR-IOV interface which is already configured using the IPAM settings provided for the object.

```
# cat sriov-test.yaml
apiVersion: v1
kind: Pod
metadata:
  generateName: sriov-pod-
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
      { "name": "flannel-networkobj" },
      { "name": "sriov-net" }
    ]'
spec: # specification of the pod's contents
tolerations:
- operator: "Exists"
containers:
- name: multus-multi-net-poc
  image: busybox
  command: ["top"]
  stdin: true
tty: true
resources:
  requests:
```
Reference Architecture | Container Bare Metal for 2nd Generation Intel® Xeon® Scalable Processor

2. Create two multiple network-based pods from the master node using sriov-test.yaml:

   # kubectl create -f sriov-test.yaml -f sriov-test.yaml
   pod/sriov-pod-x7g49 created
   pod/sriov-pod-qk24t created

3. Retrieve the details of the running pod from the master:

   # kubectl get pods

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sriov-pod-qk24t</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>5h1m</td>
</tr>
<tr>
<td>sriov-pod-x7g49</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>5h1m</td>
</tr>
</tbody>
</table>

Once the configuration is complete, you can verify the pod networks are working as expected. To verify these configurations, complete the following steps:

1. Run “ifconfig” command inside the container:

   # kubectl exec sriov-pod-x7g49 -ti -- ifconfig

   eth0      Link encap:Ethernet  HWaddr 0A:58:0A:E9:44:12
     inet addr:10.233.68.18  Bcast:0.0.0.0  Mask:255.255.255.0
     UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
     RX packets:13 errors:0 dropped:0 overruns:0 frame:0
     TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
     collisions:0 txqueuelen:0
     RX bytes:983 (983.0 B)  TX bytes:0 (0.0 B)

   lo        Link encap:Local Loopback
     inet addr:127.0.0.1  Mask:255.0.0.0
     UP LOOPBACK RUNNING  MTU:65536  Metric:1
     RX packets:0 errors:0 dropped:0 overruns:0 frame:0
     TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
     collisions:0 txqueuelen:1000
     RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

   net1      Link encap:Ethernet  HWaddr 0A:58:0A:E9:44:02
     inet addr:10.233.68.2  Bcast:0.0.0.0  Mask:255.255.255.0
     UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
     RX packets:13 errors:0 dropped:0 overruns:0 frame:0
     TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
     collisions:0 txqueuelen:0
     RX bytes:983 (983.0 B)  TX bytes:0 (0.0 B)

   net2      Link encap:Ethernet  HWaddr 72:DD:4B:F6:C1:30
     inet addr:192.168.1.40  Bcast:0.0.0.0  Mask:255.255.255.0
     UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
     RX packets:13 errors:0 dropped:0 overruns:0 frame:0
     TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
     collisions:0 txqueuelen:1000
     RX bytes:2672232 (254.7 MiB)  TX bytes:908476 (853.6 MiB)

   # kubectl exec sriov-pod-qk24t -ti -- ifconfig
eth0 | Link encap:Ethernet  HWaddr 0A:58:0A:E9:43:0F
inet addr:10.233.67.15  Bcast:0.0.0.0  Mask:255.255.255.0
UP  BROADCAST  RUNNING  MULTICAST  MTU:1450  Metric:1
RX packets:12 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:824 (824.0 B)  TX bytes:0 (0.0 B)

lo | Link encap:Local Loopback
inet addr:127.0.0.1  Mask:255.0.0.0
UP  LOOPBACK  RUNNING  MTU:65536  Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

net1 | Link encap:Ethernet  HWaddr 0A:58:0A:E9:43:04
inet addr:10.233.67.4  Bcast:0.0.0.0  Mask:255.255.255.0
UP  BROADCAST  RUNNING  MULTICAST  MTU:1450  Metric:1
RX packets:12 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:824 (824.0 B)  TX bytes:0 (0.0 B)

net2 | Link encap:Ethernet  HWaddr 5A:33:6A:C5:B9:7D
inet addr:192.168.1.42  Bcast:0.0.0.0  Mask:255.255.255.0
UP  BROADCAST  RUNNING  MULTICAST  MTU:1500  Metric:1
RX packets:1937001 errors:0 dropped:0 overruns:0 frame:0
TX packets:756208 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:634654278 (605.2 MiB)  TX bytes:125495488 (119.6 MiB)

In the above output, 'net1' is the flannel interface (each pod is on a separate subnet), and 'net2' is the SR-IOV interface (shared subnet). This provides separation of the external access, while still allowing pods to communicate on a shared subnet.

2. Verify the networks by pinging from one pod to another through the SR-IOV (net2) interface:

```
# kubectl exec sriov-pod-x7g49 -ti -- ping 192.168.1.42
PING 192.168.1.42 (192.168.1.42): 56 data bytes
64 bytes from 192.168.1.42: seq=0 ttl=64 time=0.131 ms
64 bytes from 192.168.1.42: seq=1 ttl=64 time=0.102 ms
64 bytes from 192.168.1.42: seq=2 ttl=64 time=0.124 ms
64 bytes from 192.168.1.42: seq=3 ttl=64 time=0.113 ms
64 bytes from 192.168.1.42: seq=4 ttl=64 time=0.106 ms
64 bytes from 192.168.1.42: seq=5 ttl=64 time=0.108 ms
```

8.3 Pod using Userspace CNI plugin

The Userspace CNI plugin example focuses on connectivity between pods mainly residing on the same node (intra-node), which is provided through the Userspace CNI that connects the interface to OVS or VPP using vhost-user.

8.3.1 Pod using Userspace CNI with OVS-DPDK

The following example creates a testpmd pod with two interfaces using the 'userspace-networkobj' object, which creates vhost-user interfaces that are connected to OVS. Since this is using vhost-user, the pod is also configured with a volume mount for the socket file, and a huge page memory resource.

**Note:** 1G hugepages are not available by default and need to be created using kernel boot parameters before the system boots up. Refer to Section 5.3 for details.

Here is the testpmd pod specification example file.

```
# cat testpmd.yaml
apiVersion: v1
typed: Pod
metadata:
  generateName: testpmd-
  annotations:
    k8s.v1.cni.cncf.io/networks: userspace-networkobj, userspace-networkobj
spec:
  containers:
    - name: testpmd
      image: testpmd:v1.0
      imagePullPolicy: IfNotPresent
```
1. Create the testpod pod from the master node:
   # kubectl create -f testpmd.yaml
   pod/testpmd-wfw9v created

2. Run this command to enter the testpmd container:
   # kubectl exec -ti testpmd-wfw9v bash

3. Inside the testpmd container, run these commands:
   root@ testpmd-wfw9v:/# export ID=$(/vhu/get-prefix.sh)
   root@multi-vhost-examplewfw9v:/# testpmd -d librte_pmd_virtio.so.17.11 -m 1024 -c 0xC \
     --file-prefix=testpmd --vdev=net_virtio_user0,path=/vhu/${ID}/${ID:0:12}-net1 \
     --vdev=net_virtio_user1,path=/vhu/${ID}/${ID:0:12}-net2 --no-pci -- --no-lsc-interrupt \
     --auto-start --tx-first --stats-period 1 --disable-hw-vlan

You will see output similar to the following screenshot, which means your testpmd app can send and receive data packages. Both ports are connected through the NIC, and connectivity is verified by packets being both sent and received on both ports.
You also can have two or more pods connect the OVS via Userspace CNI plugin. Here is an example using two pods, one is pktgen and one is l3fwd.

1. Create a pktgen pod specification pktgen.yaml. This is similar to the testpmd.yaml specification from above, but instead of having two interfaces this only has one.

   ```yaml
   apiVersion: v1
   kind: Pod
   metadata:
     generateName: dpdk-pktgen-
     annotations:
       k8s.v1.cni.cncf.io/networks: userspace-networkobj
   spec:
     containers:
     - name: dpdk-pktgen
       image: dpdk-pktgen:v1.0
       imagePullPolicy: IfNotPresent
       securityContext:
         privileged: true
       runAsUser: 0
       volumeMounts:
         - mountPath: /vhu/
           name: socket
         - mountPath: /dev/hugepages1G
           name: hugepage
       resources:
         requests:
           memory: 2Gi
         limits:
           hugepages-1Gi: 2Gi
       command: ["sleep", "infinity"]
     tolerations:
     - operator: "Exists"
     nodeSelector:
       kubernetes.io/hostname: node5
   volumes:
     - name: socket
       hostPath:
         path: /var/lib/cni/vhostuser/
     - name: hugepage
       emptyDir:
         medium: HugePages
   securityContext:
     runAsUser: 0
     restartPolicy: Never
   ```

2. Create l3fwd pod specification l3fwd.yaml, which uses similar resources as pktgen.yaml, but with a different name and image:

   ```yaml
   apiVersion: v1
   kind: Pod
   metadata:
     generateName: dpdk-l3fwd-
   spec:
     containers:
     - name: dpdk-l3fwd
       image: dpdk-l3fwd:v1.0
       imagePullPolicy: IfNotPresent
       securityContext:
         privileged: true
       runAsUser: 0
       volumeMounts:
         - mountPath: /vhu/
           name: socket
         - mountPath: /dev/hugepages1G
           name: hugepage
       resources:
         requests:
           memory: 2Gi
         limits:
           hugepages-1Gi: 2Gi
       command: ["sleep", "infinity"]
     tolerations:
     - operator: "Exists"
     nodeSelector:
       kubernetes.io/hostname: node5
   volumes:
     - name: socket
       hostPath:
         path: /var/lib/cni/vhostuser/
     - name: hugepage
       emptyDir:
         medium: HugePages
   securityContext:
     runAsUser: 0
     restartPolicy: Never
   ```
3. Create the pktgen pod from the master node:

   # kubectl create -f pktgen.yaml
   pod/dpdk-pktgen-knpp4 created

4. Run this command to enter the pktgen container from master mode:

   # kubectl exec -ti dpdk-pktgen-knpp4 bash

5. Run the following commands inside the pktgen container:

   root@dpdk-pktgen-knpp4:/usr/src/pktgen# export ID=$(/vhu/get-prefix.sh)
   root@dpdk-pktgen-knpp4:/usr/src/pktgen# ./app/x86_64-native-linuxapp-gcc/pktgen -l 10,11,12 --
   vdev=virtio_user0,path=/vhu/${ID}/${ID:0:12}-net1 --no-pci --socket-mem=512 --master-lcore 10 -
   -m 11:12.0 -P
   Pktgen/>

   The pktgen application launches. Among the statistics, note the pktgen source MAC address listed as ‘Src MAC Address’.

6. In another terminal, create the l3fwd pod from the master node:

   # kubectl create -f l3fwd.yaml
7. Run this command to enter the l3fwd container from master node:

```
# kubectl exec -ti dpdk-l3fwd-t4ppr bash
```

8. Inside the l3fwd container, export the port ID prefix and start the l3fwd application. Set the destination MAC address using the ‘dest’ argument. This should be the Src MAC Address previously noted from the pktgen pod (adjust cores, memory, etc. to suit your system):

```
root@dpdk-l3fwd-t4ppr:/usr/src/dpdk/examples/l3fwd/x86_64-native-linuxapp-gcc/app# export ID=$(/vhu/get-prefix.sh)
root@dpdk-l3fwd-t4ppr:/usr/src/dpdk/examples/l3fwd/x86_64-native-linuxapp-gcc/app# ./l3fwd -c 0x10 --vdev=virtio_user0,path=/vhu/$ID/$ID:0:12-net1 --no-pci --socket-mem=512 -- -p 0x1 -P --config "(0,0,4)" --eth-dest=0,:<pktgen-source-mac-add> --parse-ptype
```

The l3fwd app should start up. Among the information printed to the screen will be the ‘Address’. This is the MAC address of the l3fwd port, make note of it.

9. Back on the pktgen pod, set the destination MAC address to that of the l3fwd port:

```
Pktgen:/> set 0 dst mac <l3fwd-mac-address>
```

10. Start traffic generation:
```
Pktgen:/> start 0
```

You should see the packet counts for Tx and Rx increase, verifying that packets are being transmitted by pktgen and are being sent back via l3fwd running in the other pod.

11. To exit:
```
Pktgen:/> stop 0
Pktgen:/> quit
```

You also can use Userspace CNI with VPP; it is similar to OVS-DPDK. Learn more at:


### 8.4 Pod using CPU Pinning and Isolation in Kubernetes

To run testpmd using CPU Pinning and Isolation in Kubernetes, perform the steps below.

1. Add the following in the testpmd pod spec in previous testpmd.yaml file. You can change the core number you want to use for the pod. In following case, we just use 4 exclusive cores in this example.

```
resources:
  requests:
```

Note: If the webhook is used, then you need an additional annotation. If the webhook is not used, you need additional mount points.

For more details, refer to a sample pod spec at: https://github.com/intel/CPU-Manager-for-Kubernetes/blob/master/resources/pods/cmk-isolate-pod.yaml

2. Deploy testpmd pod and connect to it using a terminal window:
   # kubectl create -f testpmd-cmk.yaml
   pod/testpmd-cmk-pm3uu created
   # kubectl exec testpmd-cmk-pm3uu -ti bash

3. Create /etc/cm/k/use_cores.sh file with the following content:

   ```bash
   #!/bin/bash
   export CORES=`printenv CMK_CPUS_ASSIGNED`
   SUB=${ID:0:12}
   COMMAND=${@//'$CORES'/'$CORES'}
   COMMAND=${COMMAND//'$ID'/'$ID'}
   COMMAND=${COMMAND//'$SUB'/'$SUB'}
   $COMMAND
   ```

   Note: The above script uses CMK to assign the cores from temporary environment variable 'CMK_CPUS_ASSIGNED' to its local variable CORES. Then, this variable substitutes the $CORES phrase in the command provided below as argument to this script and executes it with the correct cores selected.

4. Add executable rights to the script:
   # chmod +x /etc/kcm/use_cores.sh

5. Start testpmd using use_cores.sh script:
   ```bash
   root@ testpmd-cmk-pm3uu:/# export ID=$(/vhu/get-prefix.sh)
   root@multi-vhost-examplewfw9v:/# /opt/bin/cmk isolate --conf-dir=/etc/cm/k --pool=exclusive --no-affinity /etc/cm/k/use_cores.sh 'testpmd -d librte_pmd_virtio.so.17.11 -m 1024 -l $CORES --file-prefix=testpmd --vdev=net_virtio_user0,path=/vhu/$ID/$SUB-net1 --vdev=net_virtio_user1,path=/vhu/$ID/$SUB-net2 --no-pci -- --no-lsc-interrupt --auto-start --tx-first --stats-period 1 --disable-hw-vlan'
   ```

   The testpmd has requested exclusive cores from CPU Manager for Kubernetes, which have been advertised to the workload via environment variables. The workload is using the --no-affinity option, which indicates that CPU Manager for Kubernetes has left the pinning of the cores to the application and is pinning the CPU Manager for Kubernetes assigned cores itself using the variable from the script. The testpmd can get deterministic performance due to the guaranteed CPUs exclusivity by CPU Manager for Kubernetes.

   CPU Manager for Kubernetes can be utilized along with the other Intel technology capabilities to achieve the improved network I/O, deterministic compute performance, and server platform sharing benefits offered by Intel® Xeon® Processor-based platforms.

   CPU Pinning and Isolation capability is part of the related suite of changes across the orchestration layers’ stack. These Intel technologies enable platform capabilities such as discovery, intelligent configuration and workload-placement decisions resulting in improved and deterministic application performance.

9 Conclusion – Automation Eases Reference Application Deployment

This document contains notes on installation, configuration, and use of networking and device plugin features for Kubernetes. By following this document, it is possible to set up a Kubernetes cluster and add simple configurations for some of the features provided by Intel. The playbook enables users to perform automated deployments, which decreases installation time from days to hours. The included example use cases show how the features can be consumed to provide additional functionality in both Kubernetes and the deployed pods, including but not limited to, flexible network configurations, Node Feature Discovery, and CPU pinning for exclusive access to host cores.

Intel and its partners have been working with open source communities to add new techniques and address key barriers to networking adoption in Kubernetes for containers by harnessing the power of Intel Architecture-based servers to improve configuration, manageability, deterministic performance, network throughput, service-assurance and resilience of container deployments.

We highly recommend that you take advantage of these advanced network features and device plugins in container-based NFV deployment.
No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. No product or component can be absolutely secure.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, and Xeon are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

* Other names and brands may be claimed as the property of others.

© 2019 Intel Corporation