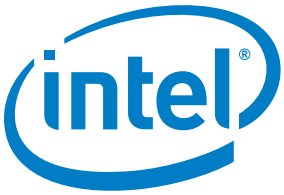


CPU Pinning and Isolation in Kubernetes*

Application Note

December 2018

Document Number: 606834-001

Contents

Figures

Tables

Revision History

Date	Revision	Description
December 2018	001	Initial release.

1.0 Introduction

This document discusses the CPU Pinning and Isolation capability, which enables efficient and deterministic workload utilization of the available CPUs. Kubernetes* supports CPU and memory first class resources, while also providing basic support for CPU Pinning and Isolation through the CPU Manager. To aid commercial adoption by adding additional CPU Management features, Intel has created CPU Manager for Kubernetes*, an open source project that introduces additional CPU utilization optimization capabilities for Kubernetes.

This document details the setup and installation of CPU Manager for Kubernetes*. It is written for developers and architects who want to integrate the new technologies into their Kubernetes-based networking solutions. CPU Manager for Kubernetes* can be utilized along with the other Enhanced Platform Awareness (EPA) capabilities in order 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 EPA methodology and the related suite of changes across the orchestration layers' stack. EPA methodology enables platform capabilities discovery, intelligent configuration and workload-placement decisions resulting in improved and deterministic application performance.

Note: This document does not describe how to set up a Kubernetes cluster. We recommend that you perform those steps as a prerequisite. For more setup and installation guidelines of a complete system, refer to the *Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors User Guide* listed in Table 2.

This document is part of the Container Experience Kit for EPA. Container Experience Kits are a collection of user guides, application notes, feature briefs, and other collateral that provide a library of best-practice documents for engineers who are developing container-based applications. Other documents in the EPA Container Experience Kit can be found at: https://networkbuilders.intel.com/network-technologies/container-experience-kits

The relevant documents include:

Document Title	Document Type
Enhanced Platform Awareness in Kubernetes	Feature Brief
Enhanced Platform Awareness in Kubernetes	Application Note
Enabling New Features with Kubernetes for NFV	White Paper
Enhanced Platform Awareness in Kubernetes	Performance Benchmark Report

2.0 Overview

Under normal circumstances, 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 applications. The positive side effect is multitasking and more efficient CPU resource utilization. The negative side effect is non-deterministic behavior which makes it unsuitable for latency sensitive workloads. A solution for optimizing these workloads performance is to 'isolate' a CPU, or a set of CPUs, from the kernel scheduler, such that it will never schedule a process thread there. 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. This results in more deterministic behavior due to reduced or eliminated thread preemption and maximizing CPU cache utilization. While beginning to guarantee the deterministic behavior of priority workloads, isolating CPUs also addresses the need to manage resources permitting multiple VNFs to coexist on the same physical server.

Prior to v1.8 of Kubernetes, CPU and Memory were the only first class resources managed by Kubernetes. CPU is requested in terms of "MilliCPU", which translates to a guaranteed time slice on a CPU effectively allowing the kernel task scheduler to act as normal. However, as mentioned above, this results in non-deterministic behavior. The Kubernetes community, Intel included, are continuing to work to develop support CPU allocation in order to provide deterministic behavior to priority workloads.

While Kubernetes continues to evolve its support for these capabilities, Intel has created an interim open source solution which is CPU Manager for Kubernetes*. CPU Manager for Kubernetes* was developed prior to the upstream CPU Manager being available, the reason behind doing this was to enable an immediate solution for CPU Pinning and Isolation in Kubernetes for commercial adoption. The gathered use cases from this solution were then used to drive the creation of the upstream CPU Manager in Kubernetes. As this still provides a limited set of features, CPU Manager for Kubernetes* is being maintained until the upstream solution is feature compatible.

3.0 Native CPU Pinning and Isolation

CPU is a compute resource in Kubernetes, which means it is a measurable quantity that can be requested, allocated and consumed. This allows users to add requests and limits for their application in the container specification of the pod. Kubernetes through the scheduler will use this information to place the pod on the most suitable node for the requests. CPU is measured in cpu units where 1 cpu is equivalent to 1 hyper-thread on a hyper-thread enabled system. CPU can be requested in fractional amounts, which means that a container requesting 0.5 of cpu will get half as much CPU time as a container requesting 1 cpu.

Prior to v1.8 of Kubernetes, there was no mechanism in Kubernetes to pin containers to CPUs or to provide any isolation guarantees. The model that was in place allowed the kernel task scheduler to move processes around as desired. The only requirement was that the workload got the requested amount of time on the CPU. So, Intel made a proposal to enhance the current model and introduce a CPU Manager component to Kubernetes. This component would have different policies of CPU Pinning and Isolation which would offer more granular assignments to the users that required it.

The CPU Manager component was introduced as an alpha feature in v1.8 of Kubernetes with the following policies:

- **None:** This policy keeps the existing model for CPU resource allocation.
- **Static:** This policy introduced CPU Pinning and Isolation at a container level to Kubernetes. With this policy enabled, a request for an integral CPU amount in a Guaranteed Pod will result in that container being allocated a whole CPU or CPU(s) with a guarantee that no other container will run on that CPU(s).

4.0 CPU Manager for Kubernetes*

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* uses a directory of folders to represent the cores available on the system. The folders in this directory are defined as pools. A pool, in this instance, is a named (e.g. exclusive) group of CPU lists. CPU Manager for Kubernetes* has three distinct pools; the exclusive, shared and infra. A pool may be exclusive, meaning only a single task may be allocated to a CPU at a time, or shared, where multiple processes may be allocated to a CPU.

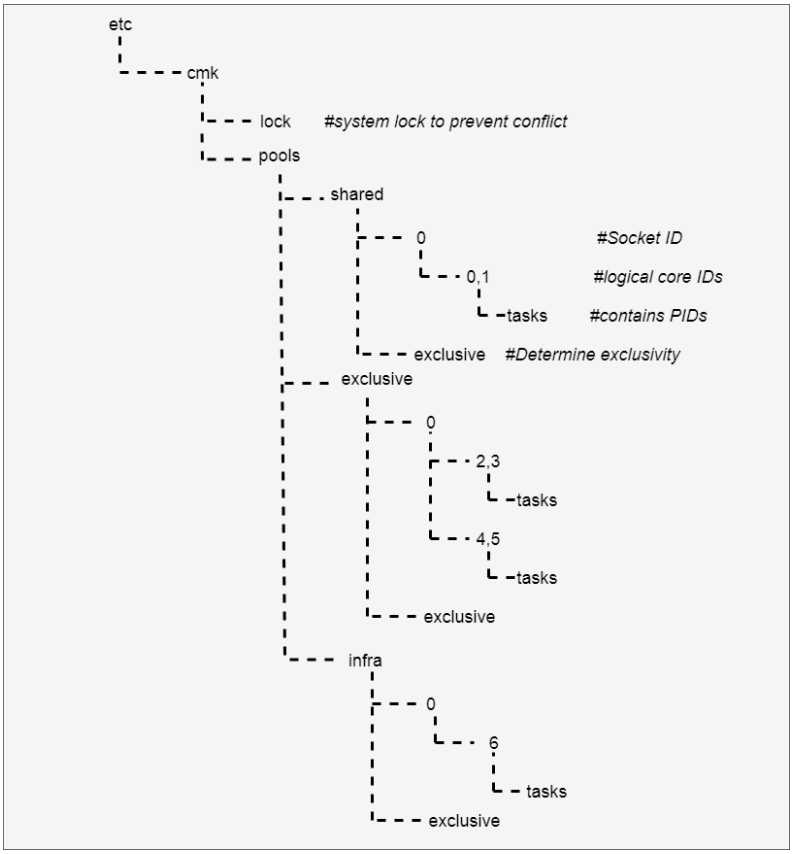
The exclusive pool within CPU Manager for Kubernetes* assigns entire physical cores exclusively to the requesting container, meaning no other container will have access to the core. To enforce this, CPU Manager for Kubernetes* advertises the number of cores available as an extended resource in Kubernetes. A user can then request a slot via the extended resources. A slot is an exclusive CPU in the exclusive pool. The use of extended resources ensures that the Kubernetes scheduler can accurately account for the exclusive slots on a node and schedule pods to appropriate nodes. The shared and infra pools are shared and are not tracked by extended resources. These pools may be requested via the command line, which includes a number of command line arguments that provide different functionality. See Section 4.3 for command line details.

When a process makes a request for a CPU, CPU Manager for Kubernetes* affinitizes the requesting process to a core on the desired pool. Along with this, CPU Manager for Kubernetes* writes the process ID (PID) to a task file on the allocated CPU. Every CPU list has an associated task file. In the case of an exclusive pool, a PID in the tasks file indicates that a core is already allocated to a workload. CPU Manager for Kubernetes* uses garbage collection to ensure that the tasks files are free from dead processes.

For ease of use, CPU Manager for Kubernetes* deploys a mutating admission webhook server. The purpose of this webhook is to add required details to a pod requesting to use CPU Manager for Kubernetes*. This removes the need for the user to be aware of what is needed exactly to run CPU Manager for Kubernetes* with their application.

CPU Manager for Kubernetes* provides ‘node-reports’ and ‘reconcile-reports’ that provide a view of the CPU Manager for Kubernetes* directory in its current state and the dead PIDs that the garbage collection has removed.

Figure 1. System State Directory for CPU Manager for Kubernetes



4.1 Installation

1. Installing CPU Manager for Kubernetes* starts with cloning the following Intel Github link:

```
#git clone https://github.com/intel/CPU-Manager-for-Kubernetes
```

2. From inside the cloned repository, the CPU Manager for Kubernetes* Docker images is built:

```
#cd <pathtorepo>
#make
```

Note: The CPU Manager for Kubernetes* image needs to be available on each node in the Kubernetes cluster where CPU Manager for Kubernetes* will be deployed.

3. CPU Manager for Kubernetes* uses RBAC and service accounts for authorization. Deploy the following yaml files:

```
#kubectl create –f cmk-rbac-rules.yaml
#kubectl create –f cmk-serviceaccount.yaml
```

4. Use the isolcpus boot parameter to ensure exclusive cores in CPU Manager for Kubernetes*are not affected by other system tasks:

```
#isolcpus=0,1,2,3
```

   a. On a hyper-threaded system, fully 'isolate' a core by isolating the hyper-thread siblings.
   b. At a minimum, the number of fully isolated cores should be equal to the desired number of cores in the exclusive pool.
   c. CPU Manager for Kubernetes* will work without the isolcpus set but does not guarantee isolation from system processes being scheduled on the exclusive data plane cores.

5. The recommended way to install CPU Manager for Kubernetes* is through the 'cluster-init' command deployed as part of a Kubernetes pod. 'Cluster-init' creates three additional pods on each node where CPU Manager for Kubernetes* is to be deployed. The first pod executes the init, install and discover CPU Manager for Kubernetes* commands, the second deploys a daemonset to execute and keep alive the 'nodereport' and 'reconcile' CPU Manager for Kubernetes* commands and the third creates a deployment for the mutating admission webhook. The function of each of these commands is explained in Section 4.3.

```
#kubectl create –f resources/pods/cmk-cluster-init.yaml
```

   a. 'Cluster-init' accepts a variety of command line configurations. An example 'cluster-init' command:

```
#/cmk/cmk cluster-init –all-hosts
```

   b. An example 'cluster-init' pod specification can be found at: https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes/blob/master/resources/pods/cmk-cluster-init-pod.yaml
   c. CPU Manager for Kubernetes* can be deployed through calling the CPU Manager for Kubernetes* commands individually if 'cluster-init' fails. Information on this can be found at: https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes/blob/master/docs/cli.md

4.2 Example Usage

The 'isolate' command is used to pin a workload to a core in the requested pool. CPU Manager for Kubernetes* uses the binary installed as part of the deployment process to act as a wrapper to the workload and runs before it. This allows CPU Manager for Kubernetes* to pin the parent process to the allocated core and to clean up on termination.

An example 'isolate' command requesting a core on the exclusive pool:

```
#/opt/bin/cmk isolate --conf-dir=/et/cmk --pool=exclusive sleep
inf
```

An example ‘isolate’ command for a pod requesting an exclusive core on the data plane core can be found at: https://github. com/Intel-Corp/CPU-Manager-for-Kubernetes/blob/master/resources/pods/cmk-isolate-pod.yaml

4.3 CPU Manager for Kubernetes* Commands

The full list of commands can be found on the CMK GitHub repository listed in Table 2. The following subsections provide an overview of each command.

4.3.1 Init

‘init’ is the first command run when installing CPU Manager for Kubernetes* on a Kubernetes cluster. The ‘init’ command creates the directory hierarchy for pools and slots. At a minimum, three pools are created, the exclusive, shared and infra pools. The exclusive pool is exclusive whereas the shared and infra pools are shared.

4.3.2 Install

The ‘install’ sub-command builds an executable binary for CPU Manager for Kubernetes* that will be located in the installation directory.

4.3.3 Discover

The ‘discover’ command uses Extended Resources to advertise the number of slots on the relative Kubernetes node. The number of slots advertised is equal to the number of CPU lists available under the exclusive pool. After the ‘discover’ command is run, the node will be patched with ` cmk.intel.com/exclusive-cores:'. The ‘discover’ command also taints each node that it has been installed on. This means that no pods will be scheduled on this node unless they have the appropriate toleration to the taint. Any pod that wishes to use CPU Manager for Kubernetes* must include the correct toleration in the pod specification. The ‘discover’ command will also add a label to the nodes to easily identify them as CPU Manager for Kubernetes* nodes.

4.3.4 Reconcile

The ‘reconcile’ command creates a long-living daemon that acts as a garbage collector in the event that CPU Manager for Kubernetes* fails to clean up after itself. The ‘reconcile’ command process runs periodically at a requested interval (10-second intervals, for example). At each interval, ‘reconcile’ command verifies the liveness of the process IDs attached to the tasks file. The ‘reconcile’ command process creates a report of any processes it has killed.

The ‘reconcile’ command creates a Kubernetes Custom Resource Definition (CRD) ReconcileReport and publishes these reports to the API server. The representation will

show the tasks that the 'reconcile' command process cleaned up because the CPU Manager for Kubernetes* did not correctly remove the programs.

4.3.5 Node-Report

The 'node-report' command prints a JSON representation of the CPU Manager for Kubernetes* configuration directory and its current state for all the nodes in the cluster that have CPU Manager for Kubernetes* installed. The representation will show the pools in the directory, the CPU lists of the pools, the exclusivity of the pools, and any process IDs that are currently running on the CPUs. There is an option to publish the 'node- report' to the API server as a CRD. Node reports are generated at a timed interval that is user defined – the default time is every 60 seconds.

4.3.6 Webhook

The 'webhook' command runs webhook server application which can be called by Mutating Admission Controller in the API server. When the user tries to create a pod which definition contains any container requesting the advertised Extended Resources, the webhook modifies it by injecting environmental variable and additional modifications to the pod which are defined in the mutations configuration file in YAML format. Mutations can be applied per pod or per container.

Default configuration deployed during cmk cluster-init adds the installation and configuration directories and host /proc filesystem volumes, service account, tolerations required for the pod to be scheduled on the CPU Manager for Kubernetes*enabled node and appropriately annotates pod. Containers specifications are updated with volume mounts (referencing volumes added to the pod) and environmental variable CMK_PROC_FS.

4.3.7 Isolate

The 'isolate' command consumes an available CPU from a specified pool. The 'isolate' sub-command allows a pool to be specified, in the case that the exclusive pool is specified, the Extended Resource created in the 'discover' command will be consumed. Up to the number of available cores in the pool will be consumed per container as an Extended Resource, this ensures the correct number of containers is allowed to run on a node. In the case of a shared pool, any CPU may be selected regardless of the current process allocations.

'isolate' writes its own process ID into the tasks file of the chosen core on the specified pool. This is done before executing any other commands in the container. When the process is complete, the CPU Manager for Kubernetes* program removes the process ID from the tasks file. In the case that this fails, the 'reconcile' command program will clean up any dead process IDs in the task files.

'isolate' will fail in the case where an exclusive pool is requested and there are no available CPUs left in that pool.

4.3.8 Describe

The 'describe' command prints a JSON representation of the configuration directory on a specific node and its current state. The 'describe' will show the pools in the directory, the CPU lists of the pools, the exclusivity of the pools and any process IDs that are currently running on the CPUs.

4.3.9 Cluster-Init

The 'cluster-init' command runs a set or subset of sub-commands – 'init', 'install', 'discover', 'reconcile', 'node-report' and 'webhook'. It also prepares specified nodes in a Kubernetes cluster for CPU Manager for Kubernetes*.

4.3.10 Uninstall

The 'uninstall' command removes CPU Manager for Kubernetes* from a node. The 'uninstall' process reverts the 'cluster-init' command:

- deletes reconcile-nodereport-pod-{node} if present
- removes 'NodeReport' from Kubernetes Custom Resource Definitions if present
- removes ReconcileReport from Kubernetes Custom Resource Definitions if present
- removes node label if present
- removes node taint if present
- removes node Extended Resource if present
- removes --conf-dir=<dir> if present and no processes are running that use cmk 'isolate'
- removes the binary from --install-dir=<dir>, if binary is not present then throws an error
- removes the webhook-pod along with other webhook dependencies (mutating admission configuration, secret, config map and service), if CPU Manager for Kubernetes* was installed on a cluster with mutating admission controller API.

Note: This command requires that there are no processes running that use the 'isolate' command, otherwise it will return an error and fail.

4.4 Performance Test Results using CPU Manager for Kubernetes*

The following tests demonstrate the performance of CPU Manager for Kubernetes*and Huge Pages utilization in a Kubernetes environment.

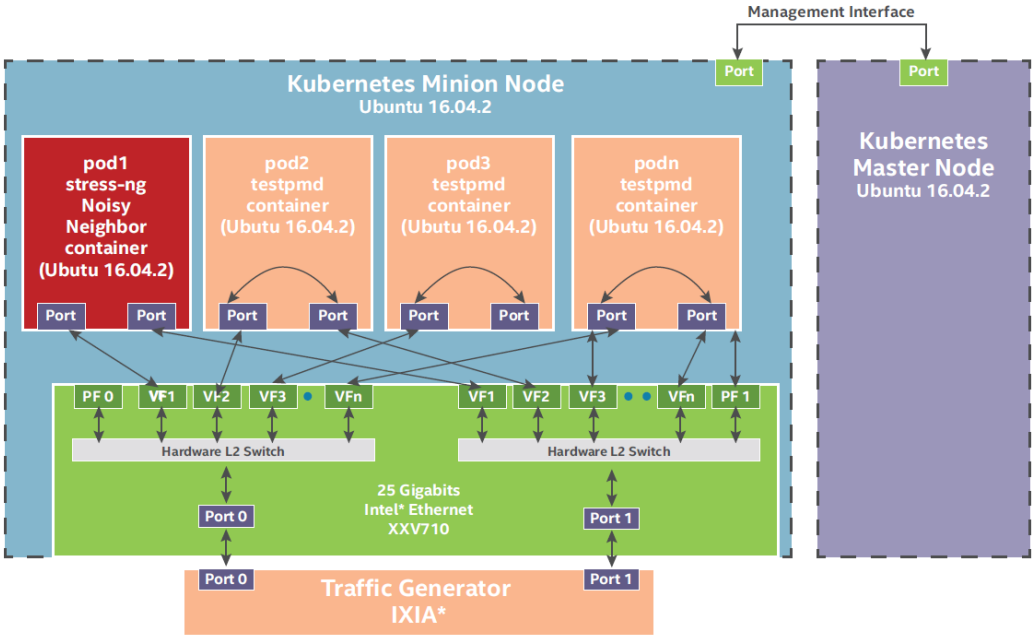
4.4.1 Test Setup

To create data flows, the tests used packet generation and testing applications that operate in userspace and in the kernel network stack. To test network throughput leveraging EPA functions, DPDK testpmd was used. Testpmd is an application that tests packet forwarding rates in virtual networks that leverage DPDK. In addition to these throughput tests, bandwidth and latency test using Qperf were also implemented. Qperf is a command line program that measures bandwidth and latency between two nodes.

4.4.2 DPDK Testpmd

The test setup for running DPDK testpmd as workload is shown in Figure 2. The traffic is generated by an Ixia traffic generator running RFC 2544. Up to 16 containers, each running the testpmd application, are instantiated using Kubernetes, with each container assigned one VF instance from each physical port on a 25G Ethernet NIC for a total of two VFs per container supporting bidirectional traffic across them. All results are tuned for zero percent packet loss. A separate container running the stress-ng application is used to simulate a noisy neighbor container.

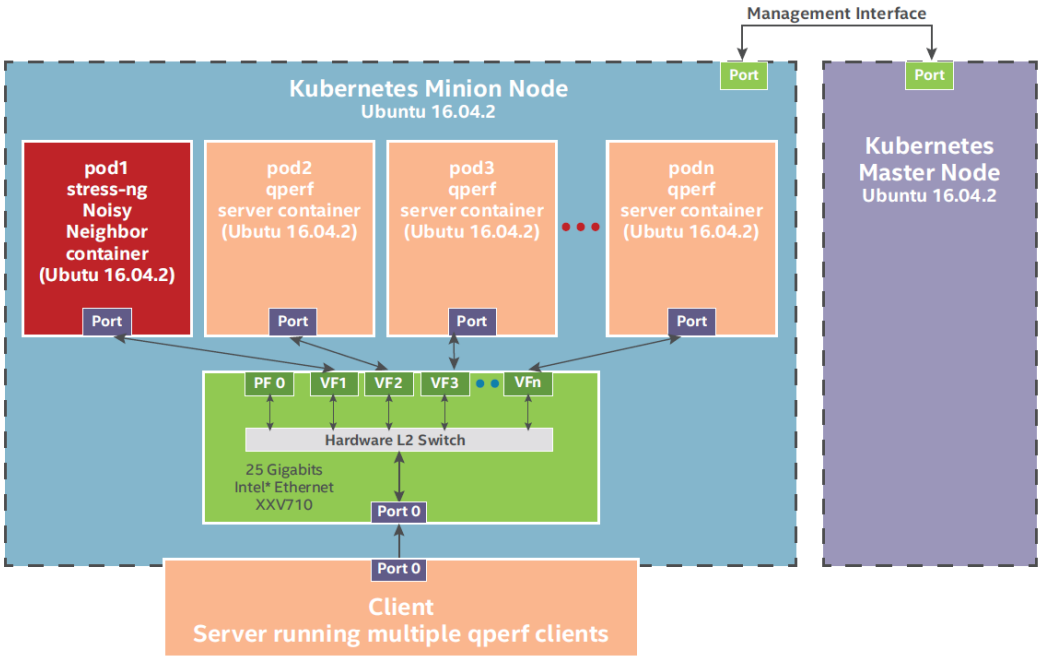
Figure 2. High-Level Overview of DPDK Testpmd Workload Setup



4.4.3 Qperf (L3 Workload)

The test setup for running qperf server workload is shown in Figure 3. The qperf clients run on a separate physical server connected to SUT using a single 25GbE NIC. Up to 16 containers, each running a qperf server, are instantiated and each are connected to one qperf client. Each container is assigned one VF instance from the same 25GbE NIC port.

Figure 3. High-Level Overview of qperf Server Workload Setup



4.4.4 Test Results

The test results with both SR-IOV VF kernel network stack and the DPDK VF stack showed consistent performance benefits with CPU Manager for Kubernetes*, which eliminates the impact of noisy neighbor applications. The test results shown in Figure 4 and Figure 5 illustrate system performance, throughput and latency, for 16 containers running the DPDK testpmd forwarding application, with and without the noisy neighbor container running in parallel, and with CPU Manager for Kubernetes* functions turned on and off.

Figure 4. DPDK Testpmd Throughput

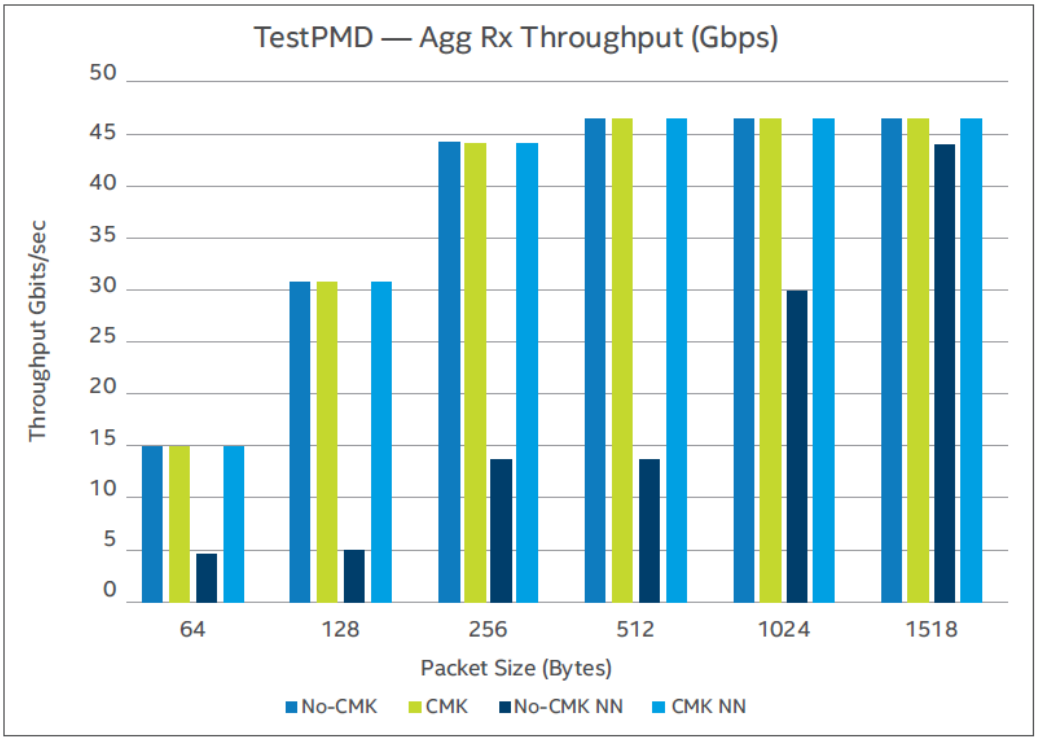
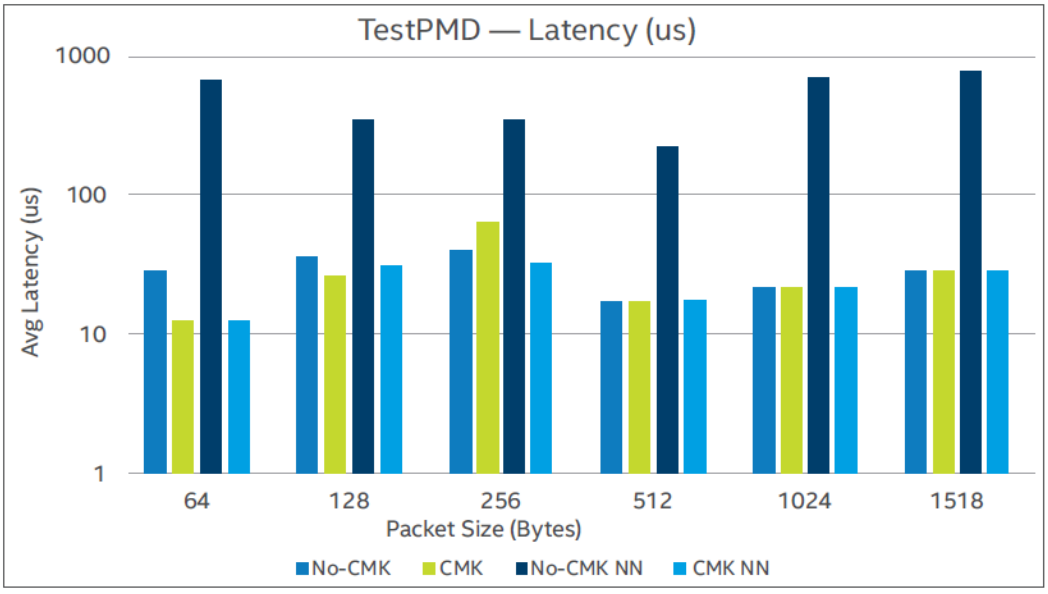


Figure 5. DPDK Testpmd Latency



Testpmd containers are deployed using Kubernetes and each container is assigned a pair of dedicated cores when using CPU Manager for Kubernetes*. CPU Manager for Kubernetes* assigns two logical cores of same physical core from exclusive pool to

each container. The DPDK Poll Mode Driver (PMD) threads in each of the testpmd containers utilize huge pages.

The results shown in Figure 4 and Figure 5 demonstrate the following:

- Without CPU Manager for Kubernetes*:
  - Performance degrades significantly in the presence of noisy neighbor.
  - Throughput decreases over 70% and latency increases by 10 times.
- With CPU Manager for Kubernetes*:
  - Performance is not impacted by having a noisy neighbor container running in the system.
  - Deterministic performance is demonstrated due to the fact that the cores are now being isolated and dedicated to the testpmd container and not shared with other containers.

The results show applications that use the DPDK network stack and utilize CMK get improved and consistent throughput and latency performance.

4.4.4.1 Qperf TCP Performance with and without CPU Manager for Kubernetes*

Test results in Figure 6 and Figure 7 show the system performance for TCP traffic tests for 16 containers running qperf server with and without noisy neighbor container present. The qperf containers are deployed using Kubernetes and qperf application is run with and without CMK. When qperf is run using CMK, two hyper-threaded sibling cores are isolated and assigned to a qperf server instance inside a container from its exclusive core pool. When qperf is run without CPU Manager for Kubernetes*, it is not pinned to any specific cores and thus is free to use any available cores in the system.

Tests are run for both TCP and UDP traffic types. Each test iteration is run for a duration of five minutes. There is one TCP connection per container for a total of 16 TCP connections for 16 containers.

Figure 6. Qperf TCP Throughput

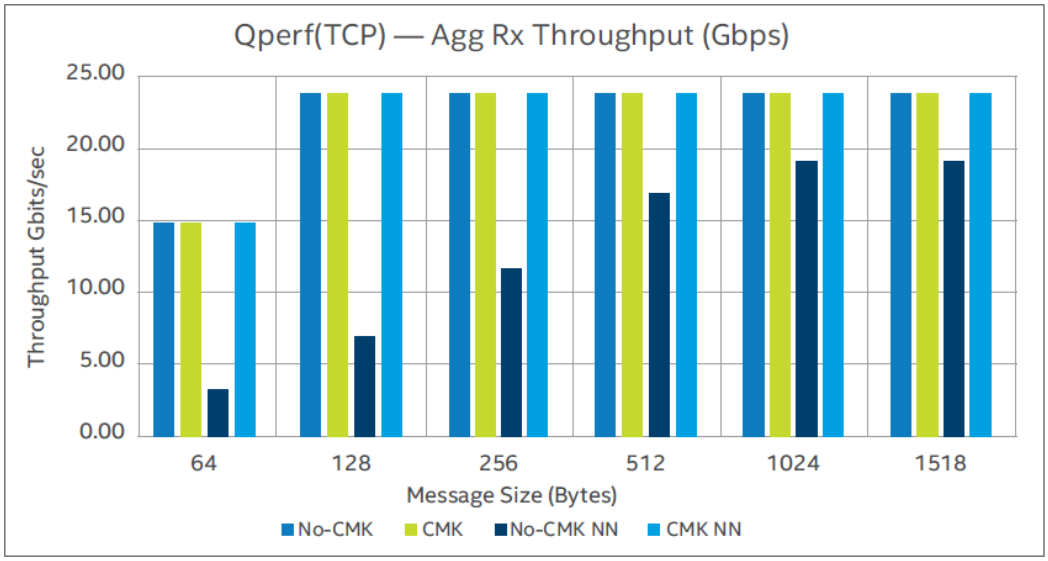
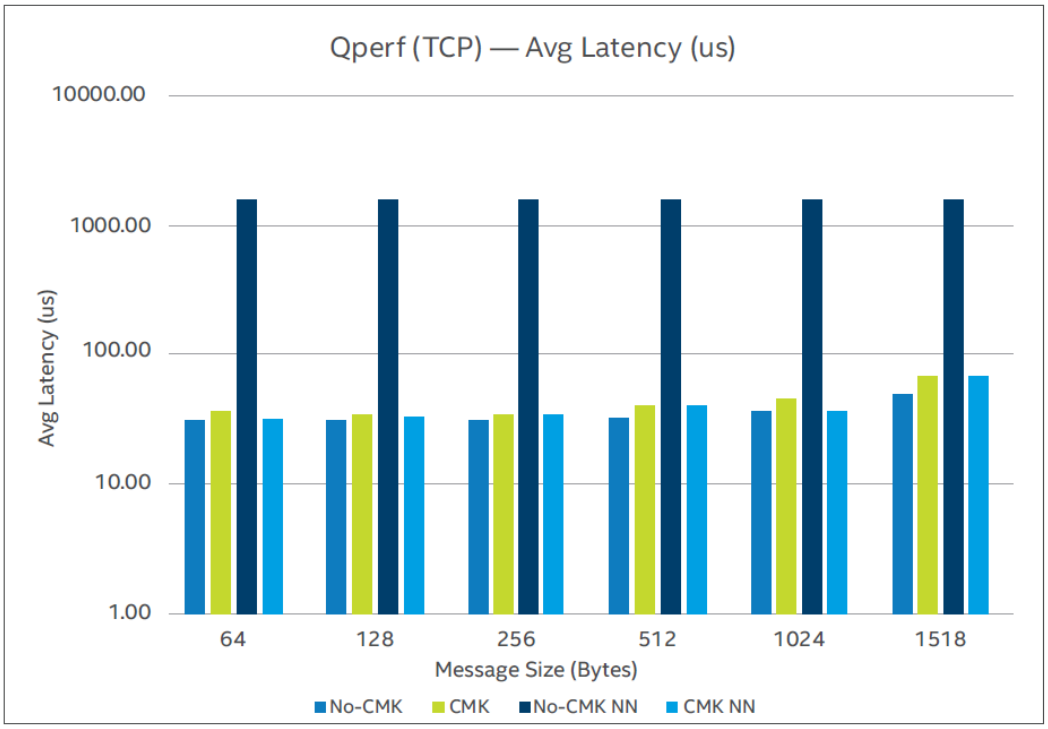


Figure 7. Qperf TCP Latency



4.4.4.2 Summary and Findings from Test Results

The results of the tests are shown in Figure 6 and Figure 7 and summarized here:

- Without CPU Manager for Kubernetes*:

  - Performance degrades significantly with noisy neighbor without CPU Manager for Kubernetes*.
  - Throughput is reduced by more than 70% for small packets and ~25% for packet sizes larger than 512 bytes. Furthermore, latency increases more than 70 times for most packet sizes.

- With CPU Manager for Kubernetes*:
  - When running the qperf server using CPU Manager for Kubernetes*, the performance is not impacted by having a noisy neighbor container running in the system.
  - The cores are now isolated and dedicated to the qperf server container and not shared with other containers, leading to a deterministic performance.

Results show the application using kernel network stack and running with CPU Manager for Kubernetes* gets consistent performance (throughput and latency).

5.0 Summary

In summary, CPU Pinning and Isolation are key requirements for applications that require deterministic performance. Intel created CPU Manager for Kubernetes* in order to enable these features with containerized deployments using Kubernetes. Intel are working with the community to bring the CPU Manager for Kubernetes* features into Kubernetes with the Static CPU Manager policy being the first step towards that.

This document also showed the benefit of CPU Pinning and Isolation in a noisy neighbor scenario. With CPU Manager for Kubernetes* enabled, the workloads performance was seen to be predictable, whereas without CPU Manager for Kubernetes*, the workload was liable to be affected by a noisy neighbor.

For in-depth details on performance benchmarks, refer to the performance benchmarking report titled *Kubernetes and Container Bare Metal Platform for NFV use cases for Intel® Xeon® Gold Processor 6138T*.

These performance benefits were showcased utilizing Intel® Xeon® Scalable servers using a sample DPDK-based user space workload and qperf workload that uses the kernel network stack, demonstrating the importance of CPU Pinning and Isolation in Kubernetes.

For more information on what Intel is doing with containers, go to https://networkbuilders.intel.com/network-technologies/ intel-container-experience-kits.

Appendix A Terminology and References

Table 1. Terminology

Term	Definition
EPA	Enhanced Platform Awareness.
Exclusive CPU	An entire physical core dedicated exclusively to the requesting container, which means no other container will have access to the core. Assigned by the exclusive pool within CPU Manager for Kubernetes*.
Exclusive pool	A group of isolated, exclusive CPUs where a container will be exclusively allocated requested amount of CPUs, meaning only that container can run on that CPU.
Pool	CPU Manager for Kubernetes* uses a directory of folders to represent the cores available on the system. The folders in this directory are defined as *pools*. A pool, in this context, is a named group of CPU lists.
Shared pool	A group of isolated, shared CPUs where a requesting container can run on any CPU in this pool with no guaranteed exclusivity.
Slot	An exclusive CPU in the exclusive pool.
Webhook server	CPU Manager for Kubernetes* deploys a mutating admission webhook server, which adds required details to a pod requesting its use.

Table 2. References

Document	Document No./Location
Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors	https://builders.intel.com/docs/networkbuilders/deploying-kubernetes-and-container-bare-metal-platform-for-nfv-use-cases-with-intel-xeon-scalable-processors-user-guide.pdf
Github repository for Intel® CPU Manager for Kubernetes	https://github.com/intel/CPU-Manager-for-Kubernetes