TECHNOLOGY GUIDE

Intel Corporation

CPU Management - CPU Pinning and Isolation in Kubernetes* Technology Guide

Authors

Philip Brownlow

Dave Cremins

1 Introduction

This document discusses the CPU pinning and isolation capability, which enables efficient and deterministic workload utilization of the available CPUs. Kubernetes* (K8s*) supports CPU and memory as first-class resources. Intel has created CPU Manager for Kubernetes* (also called CMK), an open-source project that enables additional CPU utilization optimization capabilities for K8s* and simplifies their deployment.

This document details the setup and installation of CPU Manager for Kubernetes*, set up of power management capabilities and processes for isolation, and associated performance benchmark results. The document is written for developers and architects who want to integrate the new technologies into their Kubernetes*-based networking solutions. This feature can be utilized along with the other Kubernetes capabilities in order to achieve improved network I/O, deterministic compute performance, and server platform sharing benefits offered by Intel® Xeon® Processor-based platforms.

CPU pinning and isolation is part of a set of tools developed to enable platform capabilities discovery, intelligent configuration and workload-placement decisions resulting in improved and deterministic application performance.

Note: For more setup and installation guidelines of a complete system, refer to the Deploying Kubernetes* and Container Bare Metal Platform for Network Functions Virtualization (NFV) Use Cases with Intel® Xeon® Scalable Processors User Guide listed in Table 2.

This document is part of the Container Experience Kit. 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 and can be found at: https://networkbuilders.intel.com/network-technologies/container-experience-kits

Table of Contents

Figures

Tables

Document Revision History

REVISION	DATE	DESCRIPTION
001	December 2018	Initial release of document.
002	April 2020	Added power management capabilities. Added support for exclusive-non-isolcpus pool.
003	January 2021	Added dynamic reconfiguration commands to section 3.3. Added separate dynamic reconfiguration section as 3.4. Both describe an overview of what the functionality accomplishes and how it is utilized.

1.1 Intended Audience

CPU Manager for Kubernetes* provides basic core affinity for NFV-style workloads on top of K8s. This document is intended for communication service providers who are planning and deploying virtualized mobile core infrastructure running on the latest Intel® Xeon® Scalable Processors.

1.2 Terminology

Table 1. Terminology

TERM	DESCRIPTION
CMK	CPU Manager for Kubernetes*
CPU	Central Processing Unit
CRD	Custom Resource Definition
DPDK	Data Plane Development Kit
EPA	Enhanced Platform Awareness
EPP	Energy Performance Preference. The value that associates a core with a priority level when using Intel® SST-CP.
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 number of CPUs, meaning only that container can run on that CPU.
Intel® HT	Intel® Hyper-Threading.
I/O	Input / Output
JSON*	JavaScript Object Notation
Kubernetes*	K8s*
NFV	Network Functions Virtualization
PMD	Poll Mode Driver
Pool	CPU Manager for Kubernetes* uses a Kubernetes* config-map to represent the cores available on the system. The items in this config-map are defined as pools. A pool, in this context, is a named group of CPU lists.
QoS	Quality of Service
RCU	Read Copy Update
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
SR-IOV	Single-Root Input/Output Virtualization
SST-BF	Intel® Speed Select Technology – Base Frequency
SST-CP	Intel® Speed Select Technology – Core Power
SKU	Stock Keeping Unit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VF	Virtual Function
Webhook server	CMK deploys a mutating admission webhook server, which adds required details to a pod requesting its use.

1.3 Reference Documentation

Table 2. Reference Documents

REFERENCE	SOURCE
Enhanced Platform Awareness in Kubernetes* Feature Brief	https://builders.intel.com/docs/networkbuilders/enhanced-platform-awareness-feature-brief.pdf
Enhanced Platform Awareness in Kubernetes* Application Note	https://builders.intel.com/docs/networkbuilders/enhanced-platform-awareness-in-kubernetes-application-note.pdf
Enabling New Features with Kubernetes* for NFV White Paper	https://builders.intel.com/docs/networkbuilders/enabling_new_features_in_kubernetes_for_NFV.pdf
Enhanced Platform Awareness in Kubernetes* Performance Benchmark Report	https://builders.intel.com/docs/networkbuilders/enhanced-platform-awareness-in-kubernetes-performance-benchmark-report.pdf
CPU Manager for Kubernetes* repository	https://github.com/intel/CPU-Manager-for-Kubernetes

REFERENCE	SOURCE
Intel® Speed Select Technology - Base Frequency (Intel® SST-BF) with Kubernetes* Application Note	https://builders.intel.com/docs/networkbuilders/intel-speed-select-technology-base-frequency-with-kubernetes-application-note.pdf

2 Overview

Under normal circumstances, the kernel task scheduler treats all CPUs as available for scheduling process threads and preempts executing process threads from giving CPU time to other applications. The positive side-effect of this behavior is multitasking enablement and more efficient CPU resource utilization. The negative side effect is non-deterministic performance, 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, latency-sensitive workload process threads 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 permits multiple VNFs to coexist on the same physical server.

In Kubernetes* (as of v1.18), CPU and Memory are the only first-class resources managed by the orchestration layer with the native CPU manager. 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 behavior results in non-deterministic performance. The Kubernetes* community, Intel included, is continuing to enhance support for CPU allocation in the native CPU manager to provide deterministic behavior to priority workloads. While Kubernetes* continues to evolve its support for these capabilities, Intel has created the open-source solution called CPU Manager for Kubernetes*.

2.1 CPU Manager for Kubernetes*

CPU Manager for Kubernetes* is the interim solution for CPU pinning and isolation for Kubernetes* while the native CPU Manager is being enhanced. CPU Manager for Kubernetes* contains features that the native CPU Manager does not support, specifically isolcpus. It ships with a single multi-use command-line program to perform various functions for host configuration, managing groups of CPUs, and constraining workloads to specific CPUs.

By default, CPU Manager for Kubernetes* divides up the CPUs on a system into three pools by nature/degree of isolation, with one additional optional pool. Pool types are described more detail in Section 3. The optional pool is used in cases where a user wants a process isolated from other processes on the system that cannot be placed on cores that are a subset of isolcpus. Refer to Section 3.5 for more details about this additional pool.

To isolate a process, CPU Manager for Kubernetes* uses a wrapper program, taking arguments to run the given process and sets its core affinity based on which pool it is requesting a CPU from. CPU Manager for Kubernetes* keeps track of the CPUs on a system, using a Kubernetes* config-map structure, which acts as a checkpoint for each pool. The checkpoint describes all configured pools, their options, the CPUs associated with that pool, and any tasks that are currently running on a CPU in that pool. Once a process finishes running, its process ID (PID) is removed from the corresponding task entry in the appropriate pool of the checkpoint config-map. A program constantly running in the background monitors the process IDs in each CPU task entry to make sure that there are no "zombie" processes that have died but have not been deleted from the checkpoint config-map.

An example of using CPU Manager for Kubernetes* would be if you had two high-priority processes A and B, and a third low-priority Process C. A and B need to be isolated from other processes, so they get placed in the exclusive pool on cores that are isolated using isolcpus. C is a low-priority process and is placed in the shared pool.

Reasons for a process being high priority might include:
- It may be sensitive to CPU throttling, context switches, or processor cache misses
- It benefits from sharing a processor's resources
- It requires hyper-threads from the same CPU
- It is a workload for accelerating packet processing and requires a dedicated core

Figure 1 illustrates the initial setup with no processes added. Isolcpus is equal to 0,1,2,3,4,5,6,7,12,13,14,15,16,17,18,19.

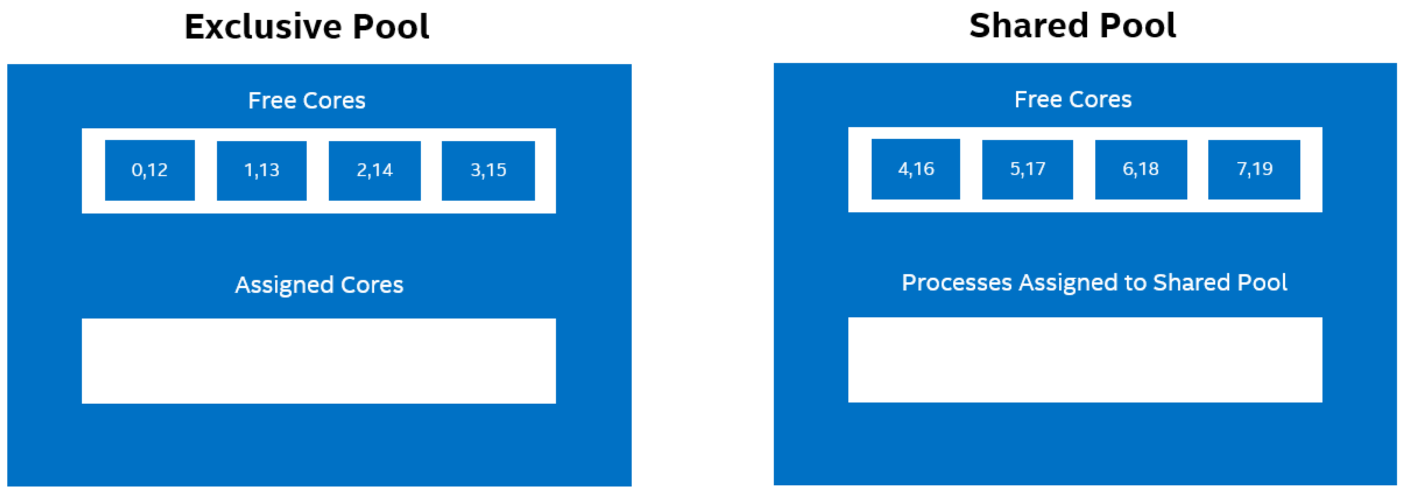

Figure 1. Initial CPU Manager for Kubernetes* Pool Configuration

Figure 2 illustrates a snapshot of pools after processes A, B, and C are added from the above scenario.

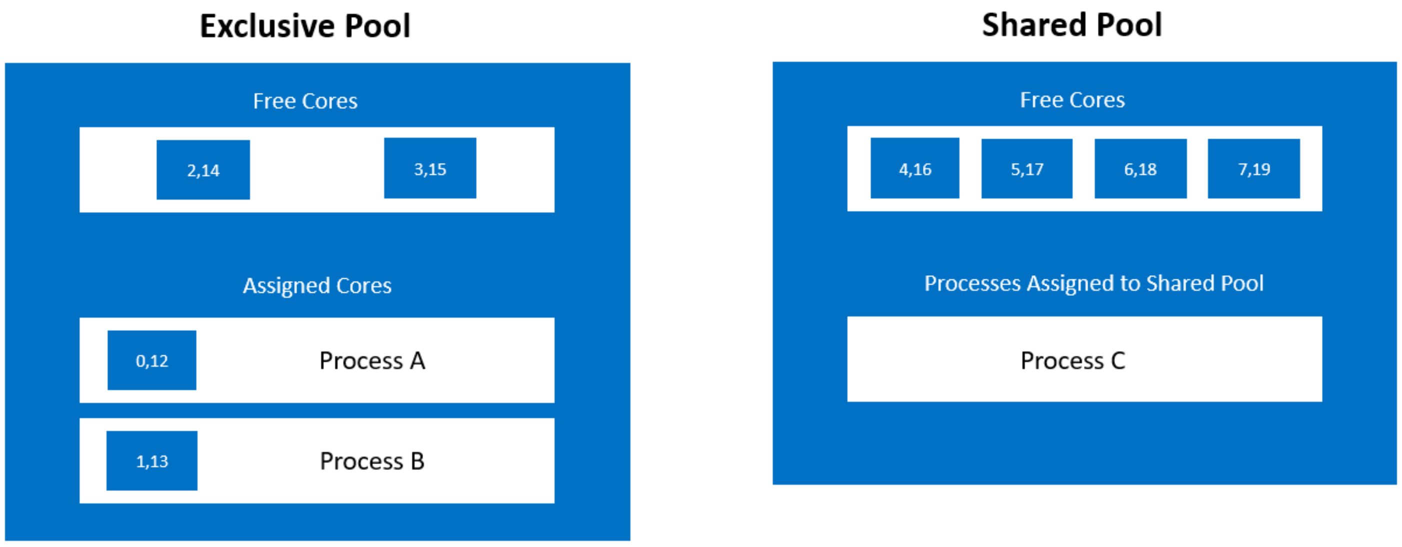

Figure 2. CPU Manager for Kubernetes* Pools with Deployment

The Infrastructure pool has not been included in the diagrams for ease of viewing. It would behave same as the shared pool and would hold the cores 8,20,9,21,10,22,11,23 that are not part of isolcpus.

2.2 CPU Manager (in native K8s)

The Native CPU offering can be enabled using the static policy for the kubelet running on your worker node. When Kubernetes* creates a pod, it assigns it to one of the following Quality of Service (QoS) classes:

- Guaranteed
- Burstable
- BestEffort

For a container to be utilized with an exclusive core, where no other container will be scheduled on the assigned core, the container must be placed in the Guaranteed QoS class, which is achieved by requesting whole numbers of CPU cores (for example, 1000m) in the pod spec. One CPU core in K8s* is equivalent to 1 hyper-thread on an Intel® Hyper-Threading (HT) Technology system. If two CPU cores are requested by a container, the CPU manager will assign both hyper-threads from a single physical core.

The kubelet allows the user to specify certain cores on which Kubernetes* processes will be placed (nicknamed “housekeeping cores”) using the *–reserved-cpus* flag. User pods that are not placed in the Guaranteed QoS class (pods that are not requesting an exclusive core) will still have access to these cores as they will be available as shared cores. The Kubernetes* processes, however, will not be placed on cores other than the ones specified. This action creates a subgroup of cores on the system that can only be utilized by user-made pods. This subgroup acts as the shared pool in a cluster as all user-made pods that are not requesting a Guaranteed QoS class have access to them. When an exclusive core is requested by a pod, the assigned core is taken out of this subgroup, so it will not be assigned to any other pods. The core is added back into this shared group when it is released.

More information about the Native CPU Manager can be found at:
https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/

Using the same example as in Section 2.1, the `-reserved-cpus` flag is set to 0,1,8,9 in a 15-core system, isolcpus is not set, and no pods have been created.

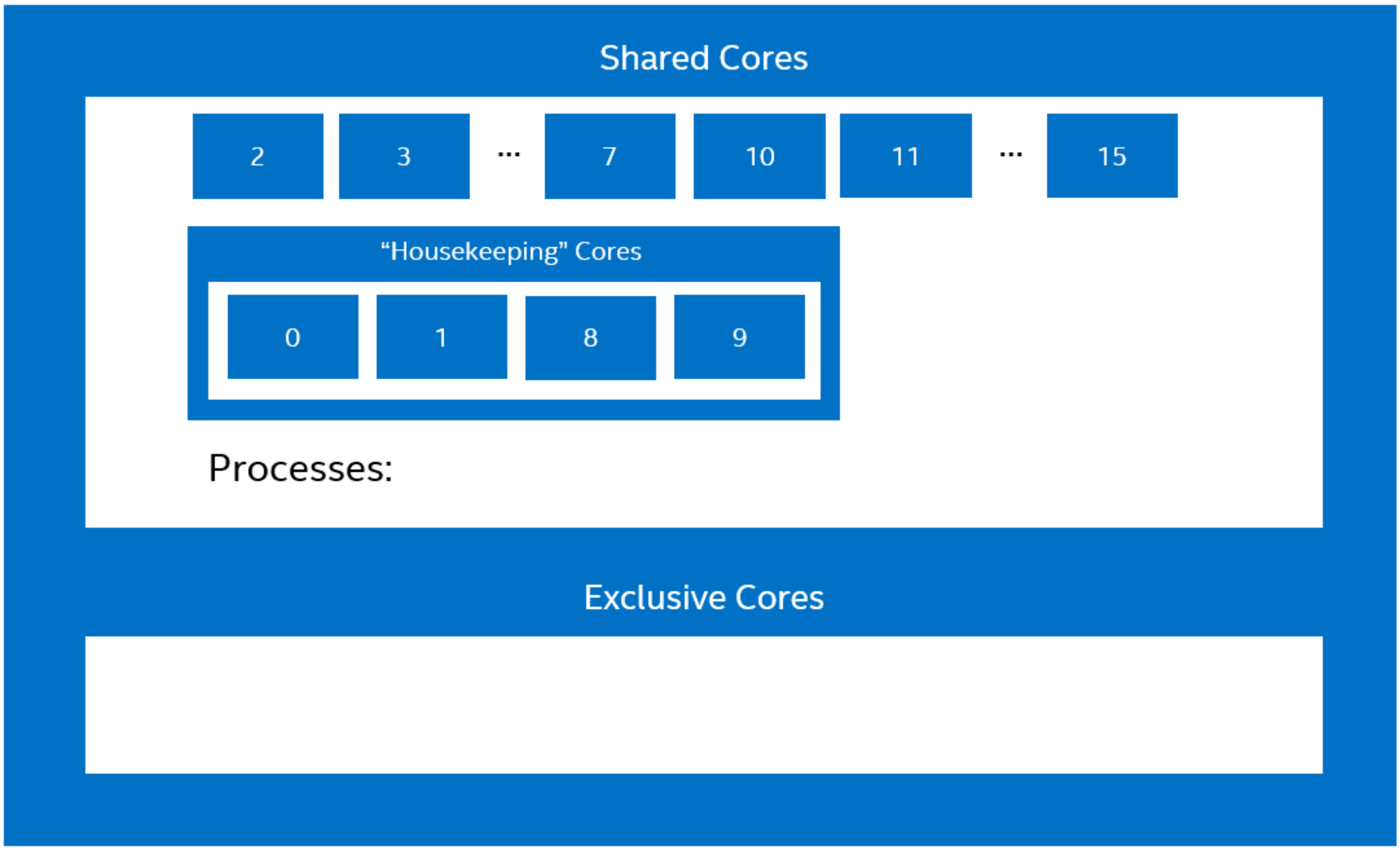

Figure 3. CPU Manager for Kubernetes* Pools: reserved-cpus Flag Set to 0,1,8,9 with No Created Pods

Now, Process A has requested 1 full core (1 hyper-thread), Process B has requested 4 full cores (4 hyper-threads), and Process C has not requested any cores.

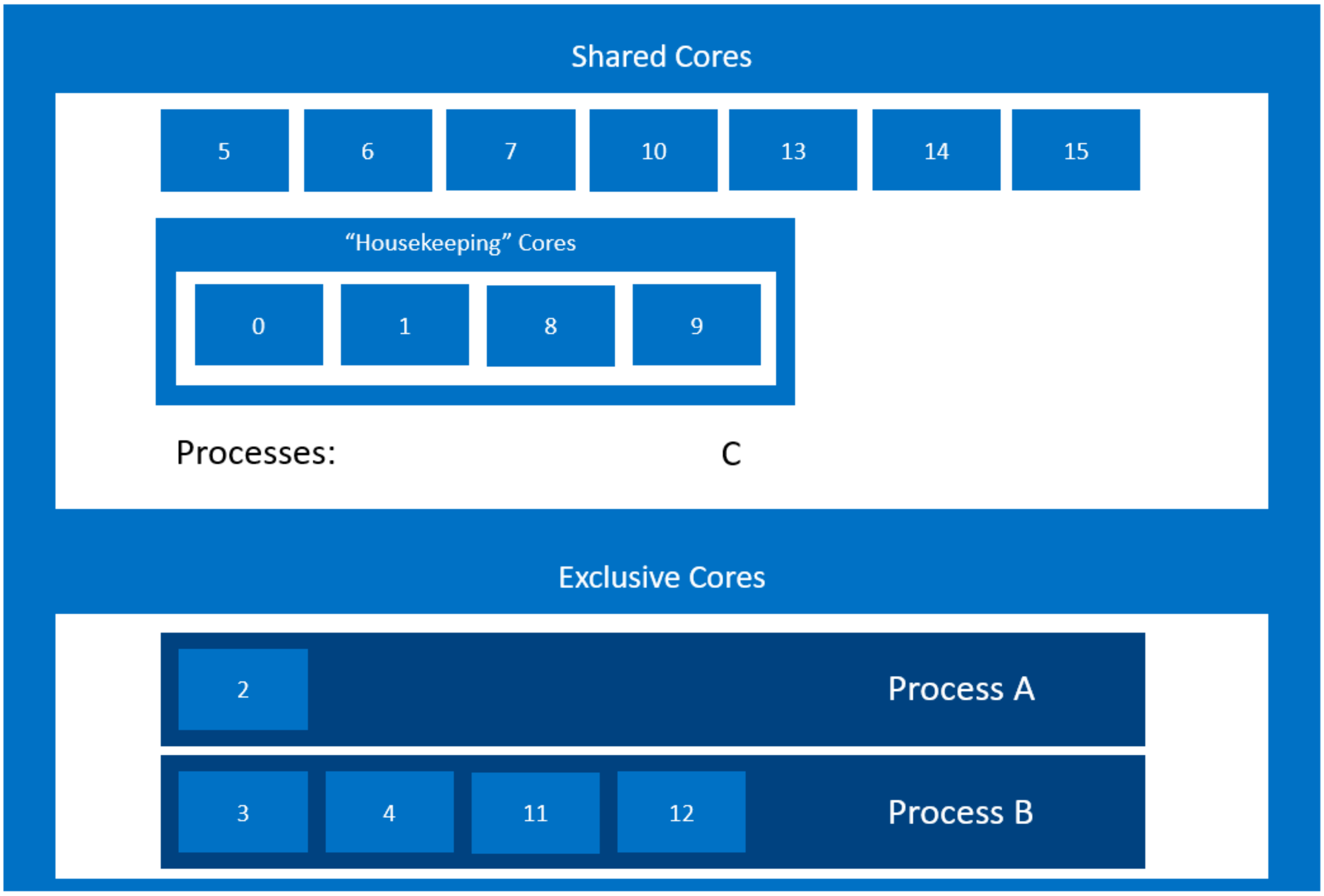

Figure 4. CPU Manager for Kubernetes* Pools with Requested Pods

Process A has been assigned core 2 and Process B has been assigned cores 3, 4, 11, and 12. All assigned cores have been taken out of the shared group, which means no other user pods can be scheduled on them. Cores 11 and 12 have been used as they are the respective hyper-thread siblings of cores 3 and 4. Cores 0, 1, 8, and 9 were not chosen as exclusive cores as they are part of the *housekeeping cores* and are only to be used by Kubernetes* processes or by pods not requesting exclusive cores.

2.3 Technology Comparison

Table 3 provides a comparison between Native CPU Manager and CPU Manager for Kubernetes.

Table 3. Technology Comparison

NATIVE CPU MANAGER	CPU MANAGER FOR KUBERNETES
K8s* code base. Beta since 1.10	Kubernetes* integration using K8s* external APIs
Updates container cgroups to provide pinning	Wrapper program that runs before workload and performs taskset command for pinning
Unaware of isolcpus	Uses isolcpus
Pod level isolation guaranteed	Gentleman's agreement for isolation
Resource account done via K8s* first class resource CPU	Resource accounting done via host file system and extended resources
Pod spec contains CPU requests	Resource accounting done via host file system and extended resources
3 CPU Pools – Shared, Reserved & Exclusive Allocations	4 CPU Pools – Exclusive, Shared, Infra & Exclusive-non-isolcpus (optional)
Shared & Exclusive pools grow and shrink dynamically as requests come in	All pools are static after deployment
NUMA Alignment with Topology Manager	NUMA alignment manual
Deployment done via K8s* release	Deployment done via a set of K8s* Pods

3 Deployment

The 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. K8s, through the scheduler, will use this information to place the pod on the most suitable node for the requests. The CPU is measured in CPU units, where 1 CPU is equivalent to 1 hyper-thread on an Intel® HT Technology system. The CPU can be requested in fractional amounts, which means that a container requesting 0.5 CPU will get half as much CPU time as a container requesting 1 CPU.

In earlier versions of K8s, 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. Intel proposed to enhance the current model and introduce a CPU Manager component to K8s to allow different policies of CPU pinning and isolation which would offer more granular assignments to the users who 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).

The remainder of this document will focus on the CPU Manager for Kubernetes* developed by Intel.

3.1 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 through 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.

Figure 3 and Figure 4 illustrate examples of core allocation. CPU Manager for Kubernetes* uses a Kubernetes* config-map to represent the cores available on the system. The entries in this config-map are defined as pools. A pool, in this instance, is a named (e.g. exclusive) group of CPU lists. CPU Manager for Kubernetes* has four distinct pools: the exclusive, shared, infra, and as of v1.4.1, exclusive-non-isolcpus. A pool may be exclusive, where 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 and exclusive-non-isolcpus pools within CPU Manager for Kubernetes* assign entire physical cores solely to the requesting container, which means no other container will have access to the core. To enforce this action, CPU Manager for Kubernetes* advertises the number of cores available as an extended resource in K8s. A user can then request a slot through the extended resources. A slot is a CPU that has been placed in either the exclusive or the exclusive-non-isolcpus pool. The use of extended resources ensures that the K8s* 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 with the command line, which includes several command-line arguments that provide different functionality. See Section 3.3 for command line details.

When a process makes a request for a CPU, CPU Manager for Kubernetes* associates the requesting process to a core on the desired pool. Along with this, CPU Manager for Kubernetes* writes the PID to a task file on the allocated CPU. Every CPU list has an associated task file. In 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 task 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*. Thus, the user need not 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 config-map in its current state and the dead PIDs that the garbage collection has removed.

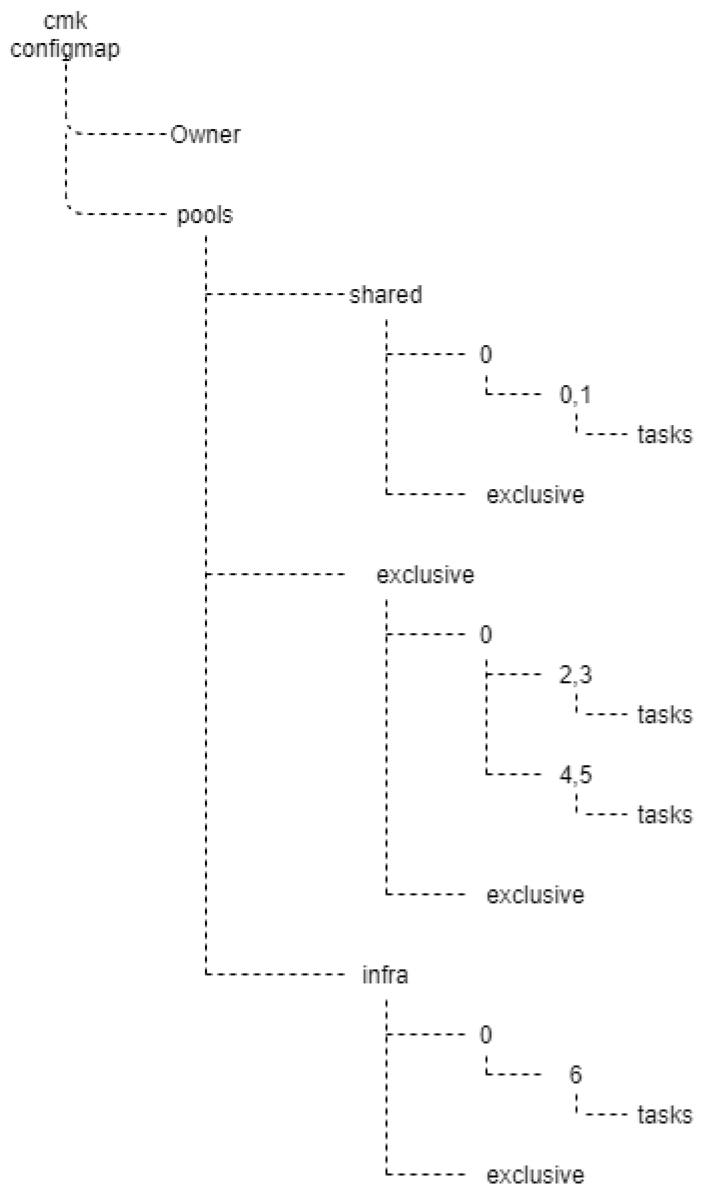

Figure 5. Deployment Diagram

3.2 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.

   CMK uses RBAC and service accounts for authorization.
3. 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 an Intel® HT Technology 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, installing and discovering the CPU Manager for Kubernetes* commands. The second deploys a daemonset to execute and to keep alive the "nodereport" and "reconcile" the 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 3.1.
```
#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 `cmk` 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
   d. See Section 3.5 for information on configuring CPU Manager for Kubernetes* with the additional exclusive-non-isolcpus pool.

3.3 CPU Manager for Kubernetes* Commands

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

3.3.1 Init

`init` is the first command run when installing CPU Manager for Kubernetes* on a K8s* cluster. The `init` command creates the config-map 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. If the CPU Manager for Kubernetes* is being configured to use the exclusive-non-isolcpus pool, then that pool will also be created by `init`, and like the exclusive pool, is exclusive.

3.3.2 Install

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

3.3.3 Discover

The *`discover`* command uses Extended Resources to advertise the number of slots on the relative K8s* 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.

As of CPU Manager for Kubernetes* v1.4.1, the `discover` command will also advertise the number of slots available in relation to the CPU lists available under the exclusive-non-isolcpus pool using Extended Resources. If CPU Manager for Kubernetes* has been configured to use the exclusive-non-isolcpus pool, the `discover` command will patch the node with *`cmk.intel.com/exclusive-non-isolcpus-cores`*. More information on the exclusive-non-isolcpus pool can be found in Section 3.5.

3.3.4 Reconcile

The *`reconcile`* command creates a long-living daemon that acts as a garbage collector if 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` 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) Reconcile Report and publishes these reports to the API server. The representation will show the tasks that the `reconcile` command process cleaned up because the CMK did not correctly remove the programs.

3.3.5 Node-Report

The *`node-report`* command prints a JavaScript Object Notation (JSON*) representation of the CPU Manager for Kubernetes* configuration config-map 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 config-map, 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 `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.

3.3.6 Webhook

The *`webhook`* command runs webhook server application that 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 that are defined in the mutation's configuration file in yaml format. Mutations can be applied per pod or per container.

The 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 the pod. Containers specifications are updated with volume mounts (referencing volumes added to the pod) and environmental variable CMK_PROC_FS.

3.3.7 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 or the exclusive-non-isolcpus pool is specified, the Extended Resource created in the `discover` command for the given pool will be consumed. Up to the number of available cores in the pool will be consumed per container as an Extended Resource, which ensures the correct number of containers can 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 process is performed 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 or exclusive-non-isolcpus pool is requested and there are no available CPUs left in that pool.

3.3.8 Describe

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

3.3.9 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 K8s* cluster for CPU Manager for Kubernetes*.

3.3.10 Uninstall

The *`uninstall`* command removes the 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 CMK was installed on a cluster with mutating admission controller API.

3.3.11 Reconfigure_setup

The reconfigure_setup command kicks off the execution of reconfiguring the CPU Manager for Kubernetes* cluster to the new desired configuration. It determines the nodes in your cluster that have been designated for the use of CPU Manager for Kubernetes* and it runs the *reconfigure* command on each of them.

3.3.12 Reconfigure

The *reconfigure command* runs on each CPU Manager for Kubernetes* node in your cluster. It determines if the new desired configuration of your cluster is possible by comparing the number of cores that have processes pinned to them (in both the exclusive pool and the shared pool) and the new number of assigned cores to the exclusive and shared pools. It also reconfigures the config-map that CPU Manager for Kubernetes* uses to keep track of the cores on the given node to portray the new layout. Finally, it executes the *reaffinitize* command to update what cores the processes have been reassigned to.

3.3.13 Reaffinitize

The *reaffinitize* command is the final step in the reconfiguration process and is the command that actually manipulates the cores that a process is allowed to run on.

3.4 Dynamic Pool Reconfiguration

Dynamic reconfiguration allows you to reconfigure the pool setup of your nodes in your cluster without having to tear down CPU Manager for Kubernetes* and clean up any configuration associated with CPU Manager for Kubernetes*. The reconfigure command will look at every pod in every namespace on all the nodes in a cluster but will only reassign those pods that have been assigned cores using CPU Manager for Kubernetes*. This reduces a considerable amount of time off the operation and makes it easier to reconfigure the cores on your node. It also means that you do not have to stop any processes that are currently running in order to reconfigure, as this method will automatically reassign any processes to the new cores in the new configuration.

For example, the following setup with the arbitrary processes 2000 and 2001 assigned to the shared pool, which contains the cores 7,15:

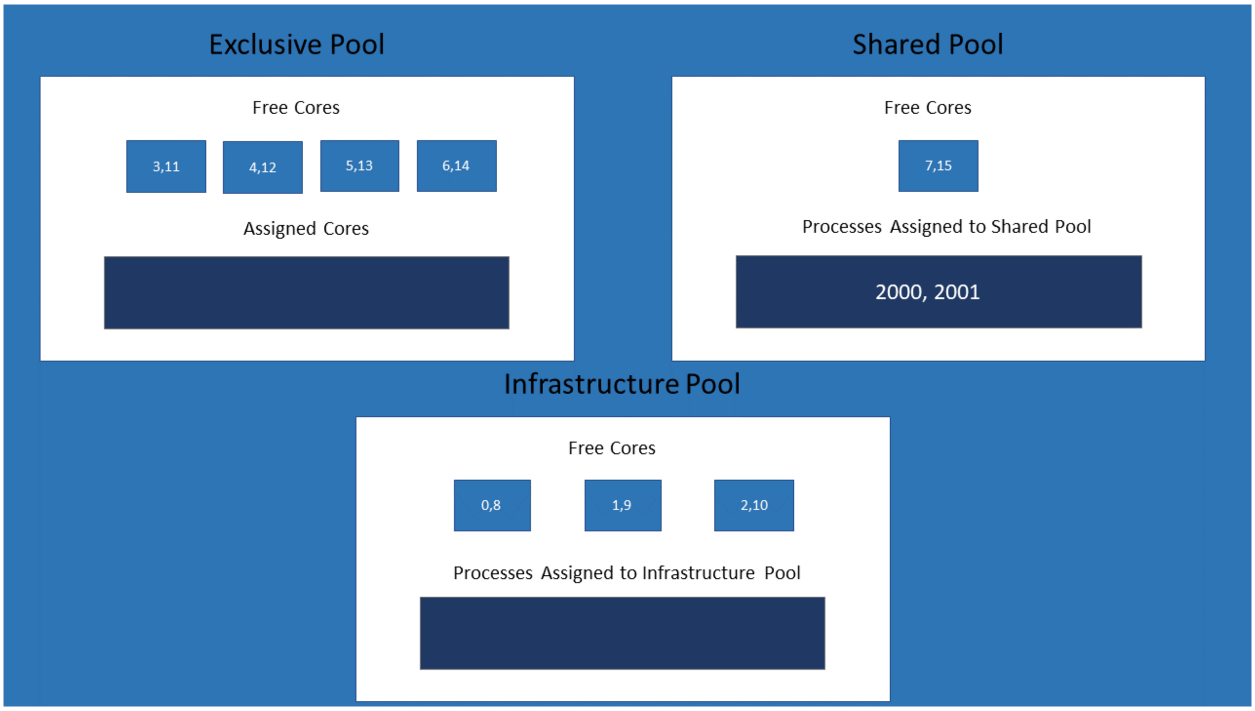

Figure 6. CPU Manager for Kubernetes* Pools – Dynamic Pool Reconfiguration

In the first screenshot, you see the initial configuration, before the reconfigure command has been executed. Here, there have been four cores assigned to the exclusive pool, and one to the shared pool. The second screenshot shows the configuration after the reconfigure command has been executed and the cores reassigned. You can see that the newly desired state is two cores assigned to the exclusive pool and three cores assigned to the shared pool. The processes 2000 and 2001 were both re-affinitized to accommodate the reconfiguration. Both processes are now able to use the additional cores 5,13 and 6,14.

In the case where a process has been pinned to a core in the exclusive pool and after reconfiguration that core is no longer in the exclusive pool, the process will be reassigned to another core that is either still in the exclusive pool after reconfiguration or has been added. If after reconfiguration the core stays in the exclusive pool, then the process is not reassigned, staying pinned to its original core, so that it does not interrupt the latency-sensitive process.

If there are not enough available cores to satisfy the newly desired configuration, for example when your configuration has three exclusive cores – all of which have a process assigned to them – and you wish to reconfigure the setup to only have two cores in the exclusive pool, the reconfigure command will recognize that one of the processes will be left unassigned and fail out before any reconfiguration has gone under way.

The reconfigure command will automatically detect which nodes in your cluster are CPU Manager for Kubernetes* nodes and it will reconfigure all of them without you having to specify. It detects the nodes by looking for the following label in the annotations of the node:

`”cmk.intel.com/cmk-node” = “true”`

This label is added by the discover operation, which occurs as part of cluster_init process, so you do not have to label the node yourself.

To use the reconfiguration command, you simply run a pod using the reconfigure_setup flag for cmk.py and pass in the configuration parameters as you would if you were setting up the cluster from scratch. For example:

`"/cmk/cmk.py reconfigure_setup --saname=cmk-serviceaccount --namespace=cmk-namespace –num-exclusive-cores=3 –num-shared-cores=2"`

3.5 Exclusive-non-isolcpus Pool

CPU Manager for Kubernetes* allows for an extra pool called the *exclusive-non-isolcpus* pool to be created using a flag passed to the init process. This pool has the exact functionality as the exclusive pool in that any cores in the exclusive-non-isolcpus pool will only run the container that the CPU Manager for Kubernetes* assigns to it. However, the difference between the two pools is that the subset of cores of isolcpus cannot be placed into the exclusive-non-isolcpus pool. The exclusive-non-isolcpus pool is also only available if there are isolated cores present on the node.

To efficiently utilize isolcpus, a process should not relinquish the core. Using the same scenario as Section 2.1, let's say that as a part of Process B, its forwarding threads use *sched_yields* (which causes the calling thread to relinquish the CPU in multithreaded programming) and RCU calls (Read Copy Update, a mechanism in Linux) to update the data structure to avoid a lock contention, which also requires a relinquish of the CPU. These calls cause a performance decrease when compared to a non-isolated setup. For this reason, it would be preferable to have it isolated from other processes on a regular core (from exclusive-non-isolcpus pool) and keep Process A isolated on an isolcpus core (from exclusive pool).

Figure 7 illustrates the initial setup of the pools when the *exclusive-non-isolcpus* pool is used, when isolcpus is equal to 0,1,4,5,6,7,12,13,16,17,18,19.

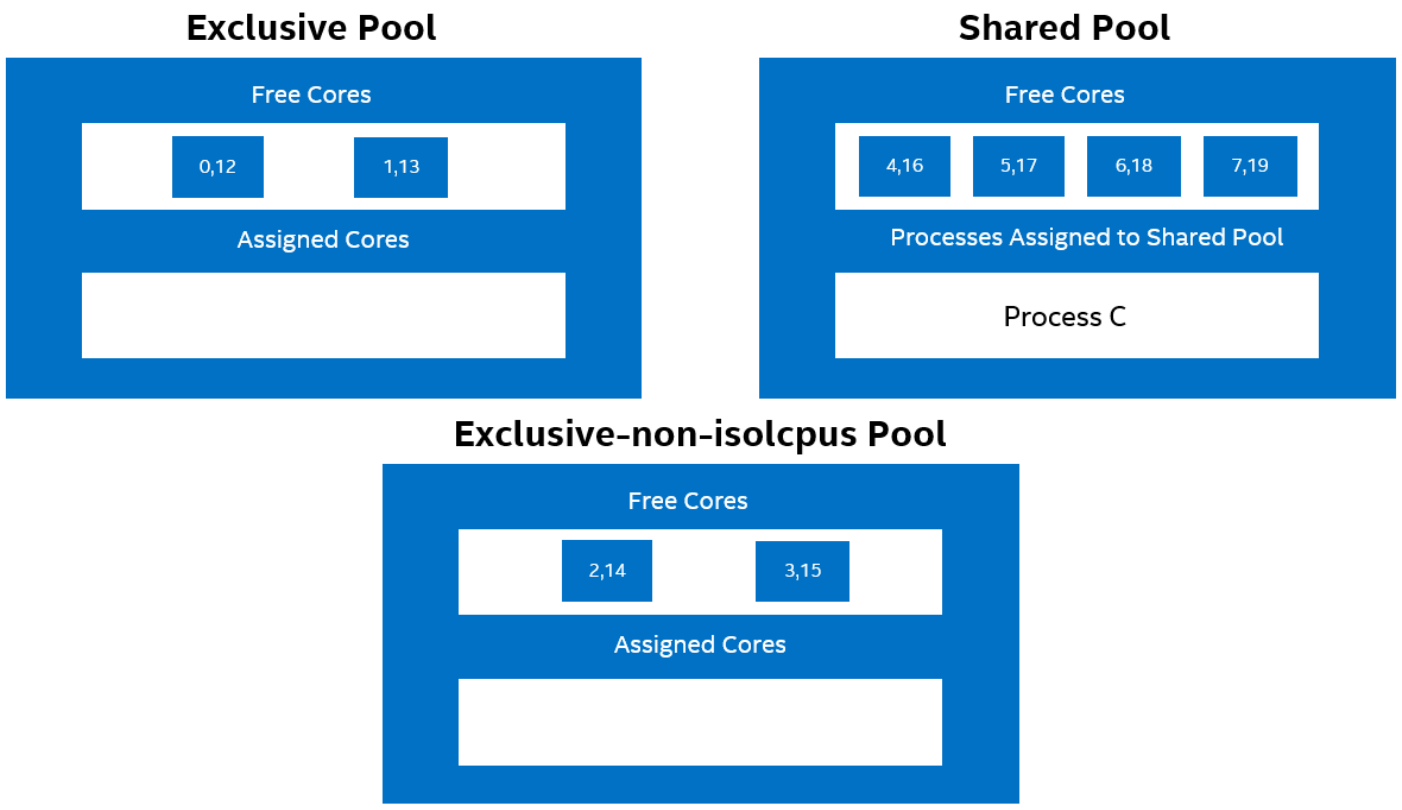

Figure 7. Initial CPU Manager for Kubernetes* Pool Configuration with Additional Pool

Figure 8 illustrates a snapshot of the pools when the processes A, B, and C are added.

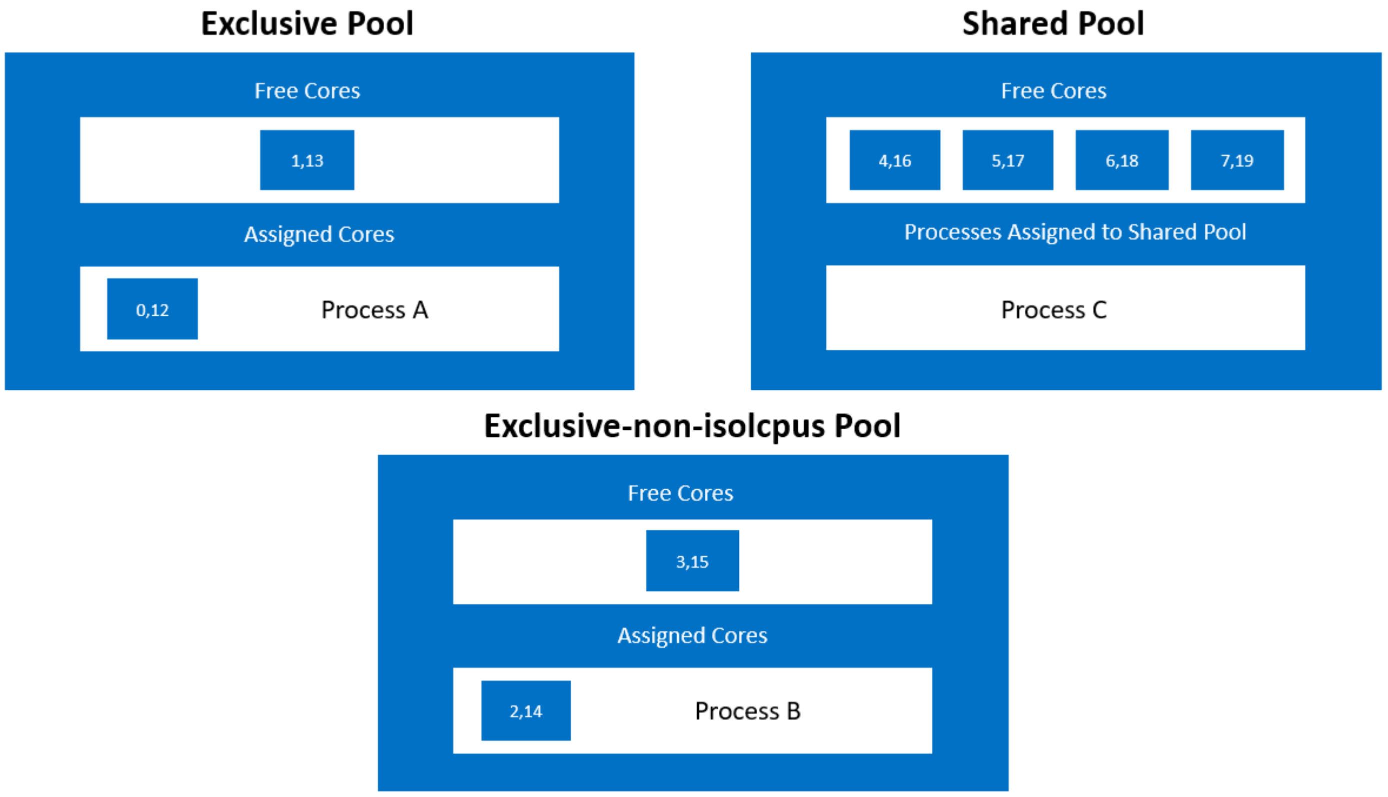

Figure 8. CPU Manager for Kubernetes* Pools with Deployment and Additional Pool

The Infrastructure pool has not been included in the diagrams for ease of viewing. It would act just as the shared pool is acting.

3.5.1 Installation

The recommended way to install CPU Manager for Kubernetes* with the exclusive-non-isolcpus pool is through the `cluster-init` command deployed as part of a K8s* pod and pass it the `-excl-non-isolcpus` flag.

For example:

```
#/cmk/cmk cluster-init --all-hosts --num-exclusive-cores=<num>            --num-shared-cores=<num>
--excl-non-isolcpus=<list>
```

The `-excl-non-isolcpus` flag takes as its parameter a list of the cores the user wants to place in the pool. The list should consist of the number of the physical cores, not the logical cores as you would use for isolcpus. Given a physical core, CMK will place both of its logical CPUs (hyper-threads) into the exclusive-non-isolcpus pool. The list can be comprised of comma-separated values, a range of values, or a mix of the two.

On a system with 32 physical cores with two logical CPUs per core:

CORE LIST	CORES PARSED
1,3,5	1,33 3,35 5,37
1-4	1,33 2,34 3,35 4,36
0,2-4,6	0,32 2,34 3,35 4,36 6,38

CPU Manager for Kubernetes* will fail if the user tries to pass as part of the parameter cores that are isolated (if they have been assigned to a pool or not), or invalid core values such as a core number that is not on the node.

3.5.2 Example Usage

The exclusive-non-isolcpus pool is used just like any other pool CPU Manager for Kubernetes* offers-– by using the `isolate` command. Specifying the exclusive-non-isolcpus pool follows the same steps as specifying the exclusive pool. The Extended Resource created for exclusive-non-isolcpus cores in `discover` will be consumed, allowing the user to consume multiple of these extended resources.

For example:

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

Request the exclusive-non-isolcpus cores using the resources section of the yaml specification for the isolate container:

```
resources:
  requests:
    cmk.intel.com/exclusive-non-isolcpus-cores: '1'
  limits:
    cmk.intel.com/exclusive-non-isolcpus-cores: '1'
```

4 Implementation Example

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-forKubernetes/blob/master/resources/pods/cmk-isolate-pod.yaml

5 Power Management Capabilities using CPU Manager for Kubernetes*

Select Stock Keeping Units (SKUs) of 2nd generation Intel® Xeon® Scalable processors include capabilities called Intel® Speed Select Technology. CPU Manager for Kubernetes* supports two of these capabilities:

- Intel® Speed Select Technology – Base Frequency (Intel® SST-BF)
- Intel® Speed Select Technology – Core Power (Intel® SST-CP.

5.1.1 Base Frequency

Intel® SST-BF is a capability that allows for certain cores to be guaranteed a base frequency, which they will never drop below. The placement of key workloads on higher frequency Intel® SST-BF cores can result in an overall system workload performance increase and overall energy savings when compared to deploying the CPU with symmetric core frequencies.

CPU Manager for Kubernetes* can be configured to place these Intel® SST-BF cores in the exclusive pool, guaranteeing that a key workload that has been pinned to an Intel® SST-BF core is isolated from other processes, and no other process can be scheduled alongside it. discovers that Intel® SST-BF is configured on a node through a label. This label is placed on the node using Node Feature Discovery, which labels the node with the *features.node.kubernetes.io/cpu-power.sst_bf.enabled* label.

Upon discovery of this label, will determine which cores on the node have Intel® SST-BF enabled and place them in the exclusive pool.

Note: For Intel® SST-BF to be configured on a node, the Intel® SST-BF cores must be isolated using the isolcpus Linux* struct. Since Intel® SST-BF cores will only be placed in the exclusive pool during setup, to utilize an Intel® SST-BF core using, request a core from the exclusive pool in the isolate yaml file.

For more information, refer to the Intel® SST-BF with Kubernetes Application Note.

5.1.2 Core Power

Intel® SST-CP is a capability that allows certain cores to be set to a higher priority level for power consumption. Cores that have a higher priority are more likely (however not guaranteed) to be supplied extra power and be pushed to turbo frequency speeds when the CPU has extra power to yield.

CPU Manager for Kubernetes* can be configured to group the cores of each priority level and place them together in the pools. The CPU Manager for Kubernetes* pools align to the priority level of the cores, so a request for a core from the exclusive pool would return affinity to a core with the highest level of priority. discovers that Intel® SST-CP is configured on the node again via a label placed on the node using Node Feature Discovery, which uses the *features.node.kubernetes.io/cpu-power.sst_cp.enabled* label.

Upon discovery of this label, CPU Manager for Kubernetes* will determine the priority level of each of the cores using their Energy Performance Preference (EPP) value, and group them together. will then determine the order of priority based on the present EPP values. In total, there are four values for priority. From highest priority to lowest priority they are:

- performance
- balance_performance
- balance_power
- power

Note: The SST technologies are disabled by default. The cores on the system have to be set up according to the technology the user wants to use, and the node has to be correctly labeled (using NFD).

will then place the cores with the highest priority EPP value in the exclusive pool, the next level of priority into the shared pool, and finally the lowest level of priority into the infra pool. If the exclusive-non-isolcpus pool is being utilized, the cores that are placed in this pool will always be from the lowest level of priority. This is because the cores with the two highest levels of priority will always be part of isolcpus, so they cannot be placed in the exclusive-non-isolcpus pool. The two scenarios in which Intel® SST-CP can be configured with CPU Manager for Kubernetes* are described below.

<u>Scenario 1 – Three EPP Values on the Node:</u>

In this scenario, the levels of priority line up with the three pools, and their placement has been highlighted above.

Table 4. Scenario 1 Example Configuration

CORES	EPP VALUE	POOL
0,2	Performance	Exclusive
1,3	Balance Performance	Shared
4-31	Balance Power	Infra

<u>Scenario 2 – Two EPP Values on the Node:</u>

In this scenario, because there are only two levels of priority, CPU Manager for Kubernetes* will divide the cores with the highest-level priority EPP value among the exclusive and shared pools based on the number of cores requested for both, and the lowest level priority cores will be placed in the infra pool.

Table 5. Scenario 2 Example Configuration, Cores and EPP Values

CORES	EPP VALUE
0-3	Performance
4-31	Balance Power

Table 6. Scenario 2 Example Configuration, Pool and Cores

POOL	CORES
Exclusive	0,1
Shared	2,3
Infra	4-31

Note: The number of isolated CPUs on the node must match the number of cores requested for the exclusive and shared pools combined.

The number of cores requested for the exclusive pool must be equal to the number of cores with the highest-level priority EPP value and similarly for the requested number of cores in the shared pool and the next level priority EPP value. When only two EPP values are present on the node, the number of requested cores for the exclusive and shared pools combined must match the number of cores with the highest-level priority EPP value.

6 Testing

This section describes the test setup that was used and the test results.[1]

6.1 Test Setup

To create data flows, the tests used packet generation and testing applications that operate in user space 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.

6.1.1 DPDK testpmd

The test setup for running DPDK testpmd as workload is shown in Figure 9. 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.

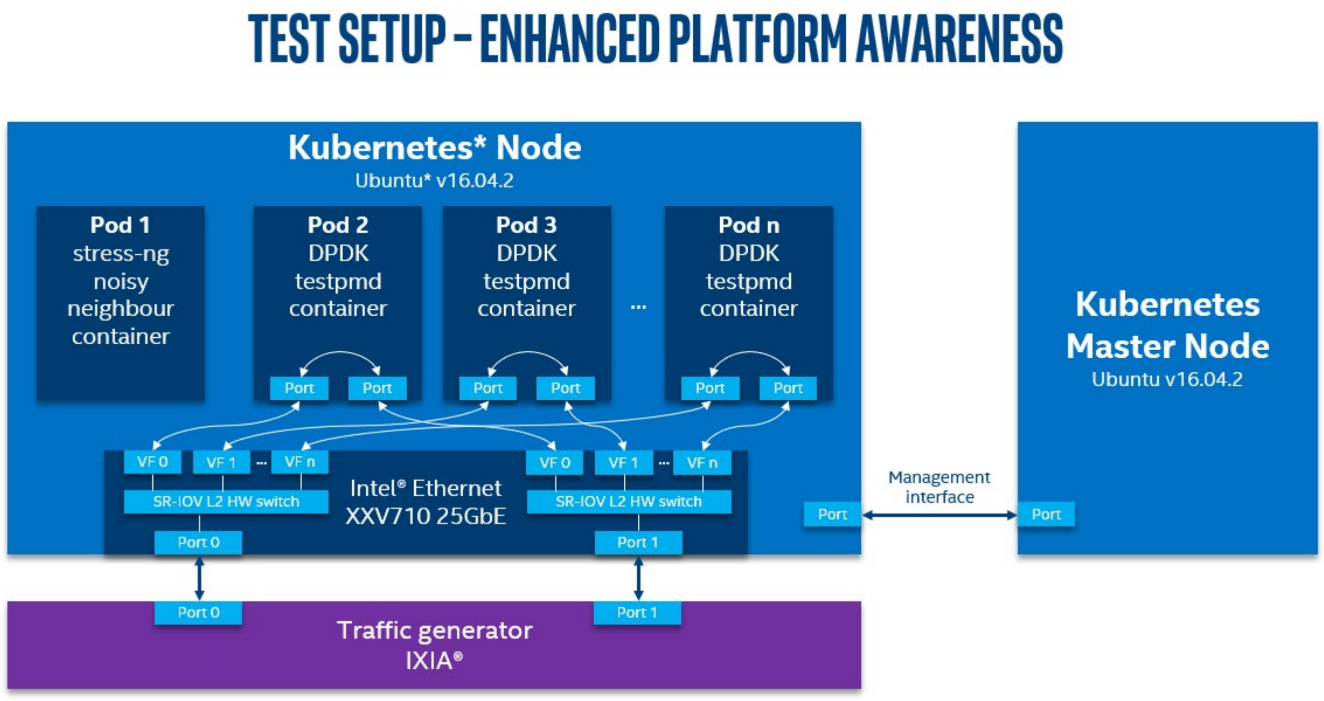
TEST SETUP – ENHANCED PLATFORM AWARENESS
Kubernetes* Node
Ubuntu* v16.04.2
Pod 1 stress-ng noisy neighbour container
Pod 2 DPDK testpmd container
Pod 3 DPDK testpmd container
...
Pod n DPDK testpmd container
Port
VF 0
VF 1
VF n
SR-IOV L2 HW switch
Intel® Ethernet XXV710 25GbE
Port 0
Port 1
Management interface
Kubernetes Master Node
Ubuntu v16.04.2
Traffic generator IXIA®

Figure 9. High-Level Overview of DPDK Testpmd Workload Setup[2]

6.1.2 Qperf (L3 Workload)

The test setup for running qperf server workload is shown in Figure 10. The qperf clients run on a separate physical server connected to the 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.

[1] Refer to http://software.intel.com/en-us/articles/optimization-notice for more information regarding performance and optimization choices in Intel software products.

[2] Refer to http://software.intel.com/en-us/articles/optimization-notice for more information regarding performance and optimization choices in Intel software products.

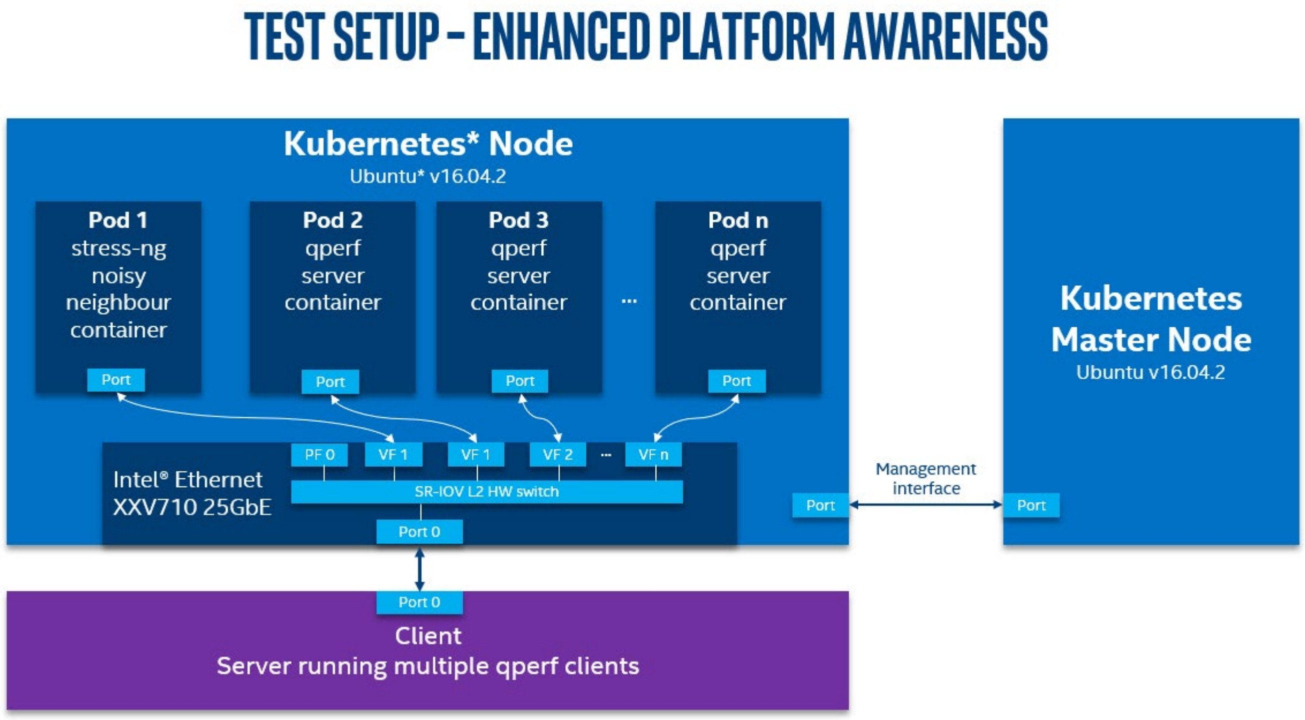
TEST SETUP – ENHANCED PLATFORM AWARENESS
Kubernetes* Node
Ubuntu* v16.04.2
Pod 1 stress-ng noisy neighbour container
Pod 2 qperf server container
Pod 3 qperf server container
Pod n qperf server container
Port
Intel® Ethernet XXV710 25GbE
PF 0
VF 1
VF 1
VF 2
VF n
SR-IOV L2 HW switch
Port 0
Management interface
Kubernetes Master Node
Ubuntu v16.04.2
Client
Server running multiple qperf clients

Figure 10. High-Level Overview of qperf Server Workload Setup

6.2 Test Results

This section describes the test results for DPDK testpmd and Qperf.

6.2.1 DPDK testpmd Performance with and without CPU Manager for Kubernetes*

The test results[3] with both SR-IOV Virtual Function (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 11 and Figure 12 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.

[3] Refer to Appendix A for configuration and date tested. See backup for workloads and configurations. Results may vary.
Refer to https://builders.intel.com/docs/networkbuilders/enhanced-platform-awareness-in-kubernetes-performance-benchmark-report.pdf for more information.

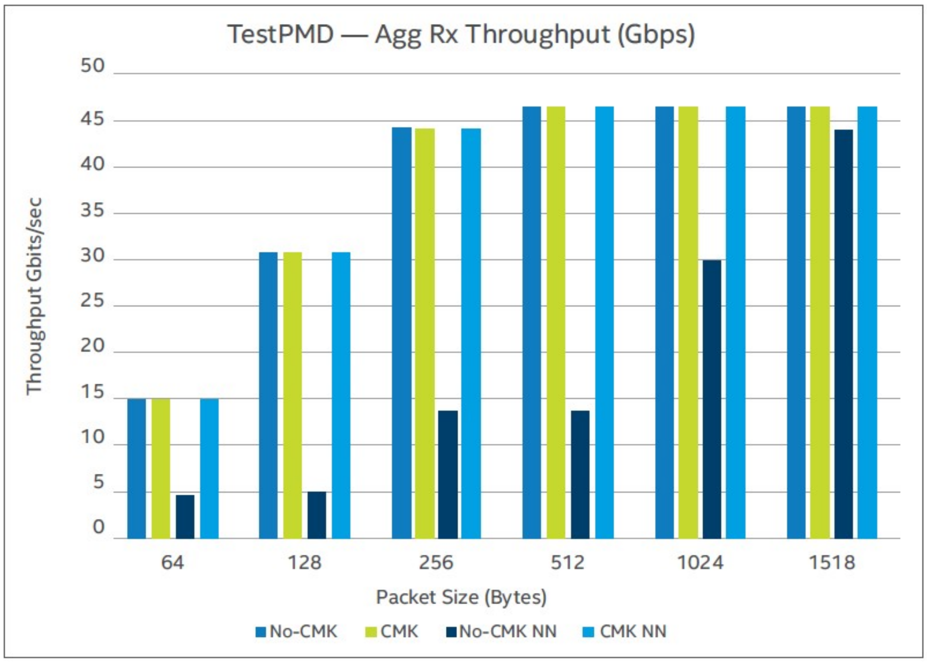

Figure 11. DPDK Testpmd Throughput (higher throughput is better)

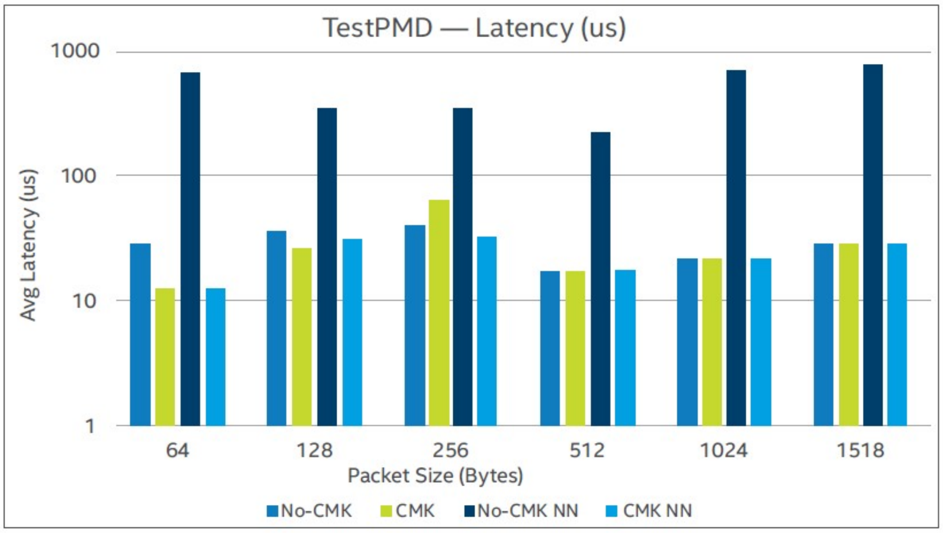

Figure 12. DPDK Testpmd Latency (lower latency is better)

Testpmd containers are deployed using K8s* 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 the 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 11 and Figure 12 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 since 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 the CPU Manager for Kubernetes* get improved and consistent throughput and latency performance.

6.2.2 Qperf Transmission Control Protocol (TCP) Performance with and without CPU Manager for Kubernetes*

Test results[4] in Figure 13 and Figure 14 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 CPU Manager for Kubernetes. When qperf is run using the CPU Manager for Kubernetes, 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 User Datagram Protocol (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.

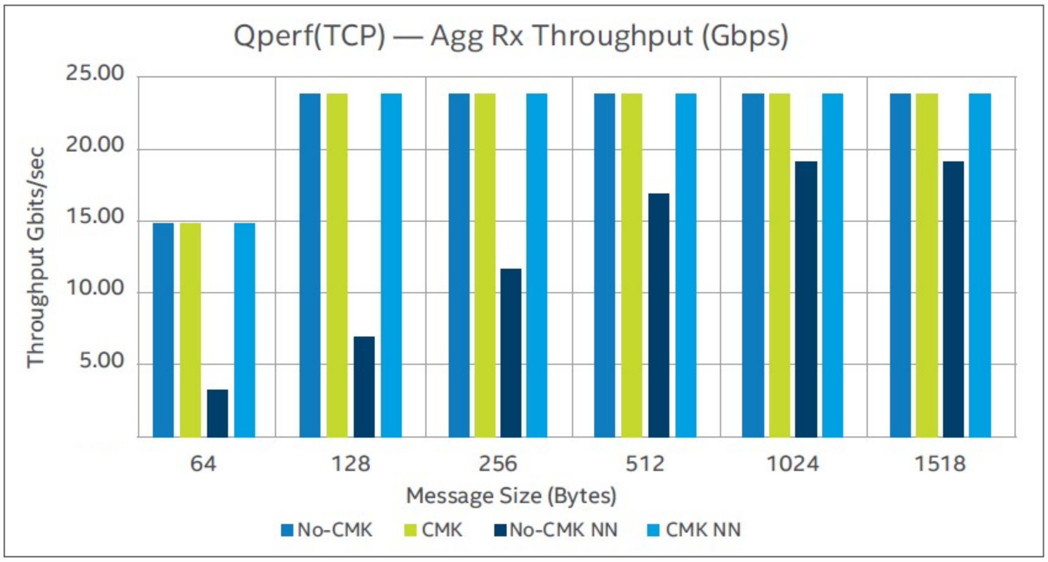

Figure 13. Qperf TCP Throughput with Noisy Neighbor Comparison (higher throughput is better)

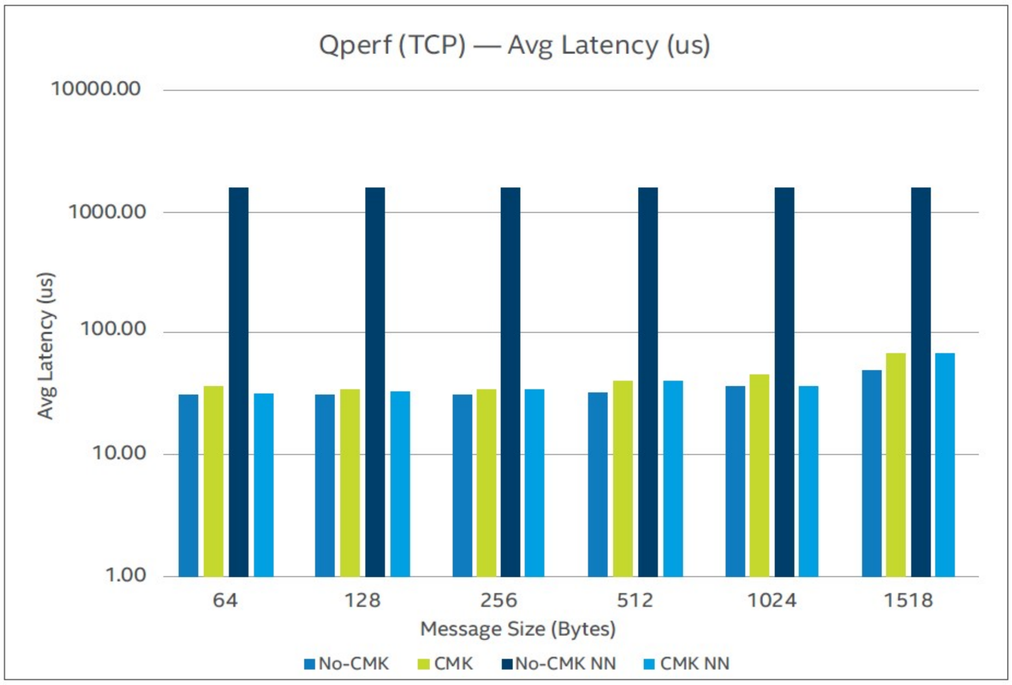

Figure 14. Qperf TCP Latency with Noisy Neighbor comparison (lower latency is better)

[4] Refer to Appendix A for configuration and date tested. See backup for workloads and configurations. Results may vary.

Refer to https://builders.intel.com/docs/networkbuilders/enhanced-platform-awareness-in-kubernetes-performance-benchmark-report.pdf for more information.

The Qperf test results[5] are shown in Figure 13 and Figure 14 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).

7 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 is working with the community to bring the CPU Manager for Kubernetes* features into Kubernetes, and the Static CPU Manager policy is 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 Enhanced Platform Awareness in Kubernetes* Performance Benchmark Report. 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/container-experience-kits.

[5] Refer to Appendix A for configuration and dates tested. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Appendix A Performance Test Configuration

The tests described in this document were performed in January 2018 to demonstrate the performance of CPU Manager for Kubernetes* and Huge Pages utilization in a K8s* environment. [6]

A.1 Hardware Configuration

Table 7. Hardware Components for Performance Benchmark Tests

ITEM	DESCRIPTION	NOTES
Platform	Intel Server Board S2600WFQ	Intel Xeon processor-based dual processor server board with 2 x 10 GbE integrated LAN ports
Processor	2x Intel Xeon Gold Processor 6138T (formerly Skylake)	2.0 GHz; 125 W; 27.5 MB cache per processor 20 cores, 40 hyper-threaded cores per Processor
Memory	192GB Total; Micron* MTA36ASF2G72PZ	12x16GB DDR4 2133MHz 16GB per channel, 6 Channels per socket
NIC	Intel Ethernet Network Adapter XXV710-DA2 (2x25G) (formerly Fortville)	2 x 1/10/25 GbE ports Firmware version 5.50
Storage	Intel DC P3700 SSDPE2MD800G4	SSDPE2MD800G4 800 GB SSD 2.5in NVMe/PCIe
BIOS	Intel Corporation SE5C620.86B.0X.01.0007.060920171037 Release Date: 06/09/2017	Hyper-Threading - Enable Boot performance Mode – Max Performance Energy Efficient Turbo – Disabled Turbo Mode - Disabled C State - Disabled P State - Disabled Intel VT-x Enabled Intel VT-d Enabled

A.2 Software Configuration

Table 8. Software Components for Performance Benchmark Tests

SOFTWARE COMPONENT	DESCRIPTION	REFERENCES
Host Operating System	Ubuntu 16.04.2 x86_64 (Server) Kernel: 4.4.0-62-generic	https://ubuntu.com/download/server
NIC Kernel Drivers	i40e v2.0.30 i40evf v2.0.30	https://sourceforge.net/projects/e1000/files/i40e%20stable/
DPDK	DPDK 17.05	http://fast.dpdk.org/rel/dpdk-17.05.tar.xz
CPU Manager for Kubernetes*	V1.0.1	https://github.com/intel/CPU-Manager-for-Kubernetes
Ansible*	Ansible 2.3.1.0	https://github.com/ansible/ansible/releases
Bare Metal Container RA scripts	Includes Ansible* scripts to deploy Kubernetes* v1.6.4 h	https://github.com/intel-onp/onp
Docker*	v1.13.1	https://docs.docker.com/engine/install/
SR-IOV-CNI	v0.2-alpha. commit ID: a2b6a7e03d8da456f3848a96c6832e6aefc968a6	https://ubuntu.com/download/server

[6] For more complete information about performance and benchmark results, visit www.intel.com/benchmarks. Refer to https://builders.intel.com/docs/networkbuilders/enhanced-platform-awareness-in-kubernetes-performance-benchmark-report.pdf

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

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.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy. Intel technologies may require enabled hardware, software or service activation.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. *Other names and brands may be claimed as the property of others.

0121/DN/WIPRO/PDF 606834-003US