Technology Guide

Kubernetes - Resource Orchestration for 4th Gen Intel® Xeon® Scalable Processors

Author

Adrian Hoban

1 Introduction

Kubernetes has become the leading software platform through which modern, cloud-native workloads are deployed. It supports an impressively varied set of use cases spanning both public cloud and private infrastructure deployments. The behavior of these use cases in terms of both the application and the infrastructure key performance indicators (KPIs) is a complex function of the application developer/deployer Kubernetes Pod specifications, the Kubernetes cluster administrator configurations at both the cluster level and the per-worker node level, the infrastructure as a service (IaaS) administrator configuration of the virtual machine (or bare metal instance) underpinning the Kubernetes worker node, and of course the physical compute, network, storage, and accelerator choice/configuration.

The implication is that while applications can be deployed unmodified in many differently configured Kubernetes environments, their KPIs will vary as will the scale-out/scale-in points that impact total cost of ownership and their sustainability behavior.

This paper explores capabilities for improving application KPIs for different use cases that can be enabled through various Kubernetes resource allocation options, configurations, and extensions. This technology guide introduces many different technologies that give users and system administrators fine grained control on placing workloads to achieve peak workload performance, optimize power consumption, optimize memory throughput, optimize block I/O, and to even optimize the cache sharing so that cloud native applications run faster, more efficiently, and with more consistent predictability.

This document is intended for technologists working in the cloud native field who are planning and deploying containerized applications running on the latest Intel® Xeon® Scalable processors.

2 Acknowledgements

This paper has been created based on the combined work of a team of technologists spanning functional areas such as architecture, engineering, project management, and product management, especially **Alexander Kanevskiy**, **Antti Kervinen**, **Atanas Atanasov**, **Brian Meaney**, **Chris MacNamara**, **Denisio Togashi**, **Derek Chilcote Bacco**, **Eero Tamminen**, **Eric Adams**, **Feruzjon Muyassarov**, **Gershon Schatzberg**, **Jukka Rissanen**, **Krisztian Litkey**, **Lukasz Danilczuk**, **Madalina Lazar**, **Matti-Pekka Sivosuo**, **Markus Lehtonen**, **Marlow Weston**, **Martin Xu**, **Michael Fu**, **Michael Kadera**, **Mikko Ylinen**, **Patricia Cahill**, **Peter Mangan**, **Philip Brownlow**, **Samu Kaajas**, **Tuomas Katila**, **Thijs Metsch**, and **Ukri Niemimuukko**. There are too many more to list here, but the extensive references section can be used to click through to the more detailed documentation and code from many of the individuals and teams involved.

The advancements noted in this paper also would not be possible without extensive collaboration with the countless people and the organizations that support them working in the Cloud Native Computing Foundation projects, and specifically in the Kubernetes community. Their on-going commitment to building a thriving ecosystem that prioritizes collaboration in the cloud native space as well as their community leadership and support are greatly appreciated.

This document is part of the Network and Edge Platform Experience Kits.

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	January 2023	Initial release.
002	February 2023	Minor updates to "Kubernetes Quality of Service", Figure 1, and Figure 4.

2.1 Terminology

Table 1. Terminology

Abbreviation	Description
ACK	Alibaba Container Service
ACPI	Advanced Configuration and Power Interface
AES-NI	Advanced Encryption Standard New Instruction
AKS	Azure Kubernetes Service
API	Application Programming Interface
AVX	Intel® Advanced Vector Extensions
AVX2	Intel® Advanced Vector Extensions 2
AVX512BW	AVX-512 Byte and Word
AVX512CD	AVX-512 Conflict Detection
BIOS	Basic Input Output System
CAT	Cache Allocation Technology
CDP	Code and Data Prioritization
CFS	Completely Fair Scheduler
CLOS	Cache Class of Service
CMK	CPU Manager for Kubernetes

Abbreviation	Description
CMT	Cache Monitoring Technology
CPU	Central Processing Unit
CRI	Container Runtime Interface
CRI-RM	Container Runtime Interface Resource Manager
C-state	Processor idle state
CXL	Compute Express Link
DDR	Double Data Rate
DLB	Intel® Dynamic Load Balancer
DSA	Intel® Data Streaming Accelerator
DSB	Death Star Bench
EKS	Elastic Kubernetes Service
EPP	Energy Performance Preference
FMA3	Fused Multiply Add 3
FPGA	Field Programmable Gate Array
GAS	GPU Aware Scheduling
GKE	Google Compute Engine
GMS	Google Micro Services (benchmark)
GPU	Graphics Processing Unit
gRPC	g Remote Procedure Call
HT	Hyper-Threading
HWP	Hardware P-state Management
IAA	Intel® In-Memory Analytics Accelerator
IaaS	Infrastructure as a Service
IDO	Intent Driven Orchestration
IOMMU	Input Output Memory Management Unit
K8s	Kubernetes
KPI	Key Performance Indicator
MBA	Memory Bandwidth Allocation
MBM	Memory Bandwidth Monitoring
NFD	Node Feature Discovery
NRI	Node Resource Interface
NUMA	Non-Uniform Memory Access
OS	Operating System
PAS	Platform Aware Scheduling
PCI	Peripheral Component Interconnect
PF	Physical Function
PPC	Platform Profile Composition
P-state	Processor Performance State
QAT	Intel® Quick Assist Technology
QoS	Quality of Service
RAS	Reliability Availability Serviceability
RDT	Intel® Resource Director Technology
RMD	Resource Management Daemon
SGX	Intel® Software Guard Extensions
SKU	Stock Keeping Unit
SNC	Sub-NUMA Clustering

Abbreviation	Description
SR-IOV	Single Root Input Output Virtualization
SST-BF	Intel® Speed Select Technology - Base Frequency
SST-CP	Intel® Speed Select Technology - Core Power
SST-TF	Intel® Speed Select Technology - Turbo Frequency
STP	Static Pools Policy
SW	Software
TAS	Telemetry Aware Scheduling
TBT	Intel® Turbo Boost Technology
TCO	Total Cost of Ownership
TSDB	Time Series Database
UEFI	Unified Extensible Firmware Interface
UFS	Uncore Frequency Scaling
UPI	Ultra Path Interconnect
USB	Universal Serial Bus
VF	Virtual Function
VMX	Virtual Machine Extension
VPU	Visual Processing Unit

3 Kubernetes

3.1 Kubernetes Quality of Service

To support the vast variety of deployments, Kubernetes provides an elegantly simplistic Quality of Service (QoS) mechanism [Ref 1] for the workloads. Three QoS classes can be associated with Kubernetes Pods, where a Kubernetes Pod is the unit of deployment that the native Kubernetes scheduler operates on. These QoS classes are defined as `BestEffort`, `burstable`, and `guaranteed`. The QoS class associated with a Kubernetes Pod is implicitly selected based on application resource requests within the Kubernetes Pod spec. The QoS class impacts Pod placement decisions through various CPU and resource allocation and management policies in the system.

The `BestEffort` QoS class implies that the operating system scheduler is responsible for CPU and memory resource sharing between Kubernetes Pods and other system services running on the worker node. This model is the least effort to consume as CPU core and memory resources do not need to be requested in the Pod spec.

The `burstable` QoS class implies that Kubernetes manages minimum (i.e., `request`) and maximum (i.e., `limit`) allocation of cores and memory to the containers within the Pod. This model allows the Pod spec author to specify a range on the compute and memory capacity that they need for each container in the Pod spec.

The `guaranteed` QoS class is associated with Pods where every container in the Pod spec specifies a whole or partial number of CPU cores where the `request` is equal to the `limit` and also specifies memory where the `request` is equal to the `limit`. Different containers can have different requirements, but in each case the `request` is equal to the `limit`.

To determine what QoS class Kubernetes has allocated to the Pod run the following command:

```
kubectl get pod <pod name> --namespace=<pod namespace> --output=yaml
```

The output has a line such as:

```
status:
  qosClass: Guaranteed
```

3.2 Kubernetes Resource Orchestration Configurations

Kubernetes provides a wide set of tuning capabilities for how it handles scheduling behavior. In Kubernetes, the scheduling is split in two phases, with the first phase focused on selecting the worker node on which to deploy the Pod, and the second phase focused on the specific resource allocation on the selected worker node to the Pod. There are cluster-wide configurations and per-worker node configurations that impact these phases. These configurations are usually described as "policies" and impact how Kubernetes deals with Pods and their containers.

3.2.1 Managed Kubernetes Resource Orchestration Configurations

In managed Kubernetes offerings a more limited set of cluster administrator configurations is available to the user. The providers of the managed Kubernetes service are responsible for most of the configuration, but there are a few important options that are typically exposed to the user. The following is a non-exhaustive list of examples from some managed Kubernetes providers:

- With the Microsoft Azure Kubernetes Service (AKS), there is a kubelet Custom Node Configuration feature. This is described by Microsoft as in technical "Preview" and allows the cluster administrator to configure CPU policy, Topology Policy, and the Linux Completely Fair Scheduler (CFS) configuration.
- The Google Kubernetes Engine (GKE) offers the cluster administrator a Node System Configuration for the kubelet that supports CPU policy selection and CFS.
- The Amazon Elastic Kubernetes Service (EKS) permits CPU manager policy configuration by the cluster administrator.
- The Alibaba Container Service for Kubernetes (ACK) with the ACK Pro Clusters supports functionality such as CPU Burst, which dynamically modifies the CPU limit applied to containers.

It is beyond the scope of this paper to articulate all cluster-wide managed Kubernetes service offerings. It is recommended that users of these services familiarize themselves with the available options and understand which of the capabilities mentioned elsewhere in this document can be used.

4 Orchestrating for Platform Capabilities

Some workloads perform optimally when they are granted access to certain platform capabilities and include the software to take advantage of these capabilities. This applies equally to cloud native and non-cloud native workloads. If an application leverages an optimization capability such as an accelerator or a CPU instruction, and that functionality is not available, then best practice is that the application software should adapt and function without it. The Node Feature Discovery capability introduced in section 4.1 presents a model for feature detection for capabilities that are classed as qualitative. Qualitative features are not allocatable in whole or in parts to Pods; instead, they are capabilities in the platform that are available to all Pods. Quantitative capabilities are allocatable in whole or part to Pods and are supported via the Kubernetes Device Plugin framework described in section 5.1.3.

There are several compelling reasons not to leave it to chance that a platform capability exists in a deployment. If an application or service administrator knows that an accelerator or optimization capability is allocated to an application deployment, the administrator can do things such as:

- Set more appropriate scaling up/down or scaling in/out points to ensure some workload KPIs are met.
- Deploy a more cost-effective application or service.
- Select a different balance of compute, network, and storage resources.
- Improve certain application/service KPIs, such as related to latency, throughput, or power.

4.1 Node Feature Discovery

In Kubernetes, the ability to match specific workload resource requests to platform capabilities is enabled with an add-on called Node Feature Discovery (NFD) [Ref 2]. NFD detects hardware features, hardware topology, and some system configurations, and advertises these within a cluster by applying labels to the worker node objects. Figure 1 shows a diagram of how NFD is deployed in a Kubernetes cluster [Ref 3]. NFD has domain-specific capability detection inclusive of CPU, kernel, memory, network, PCI, storage, system, USB, custom (for rule-based custom features), and local (for hooks for user-specific features).

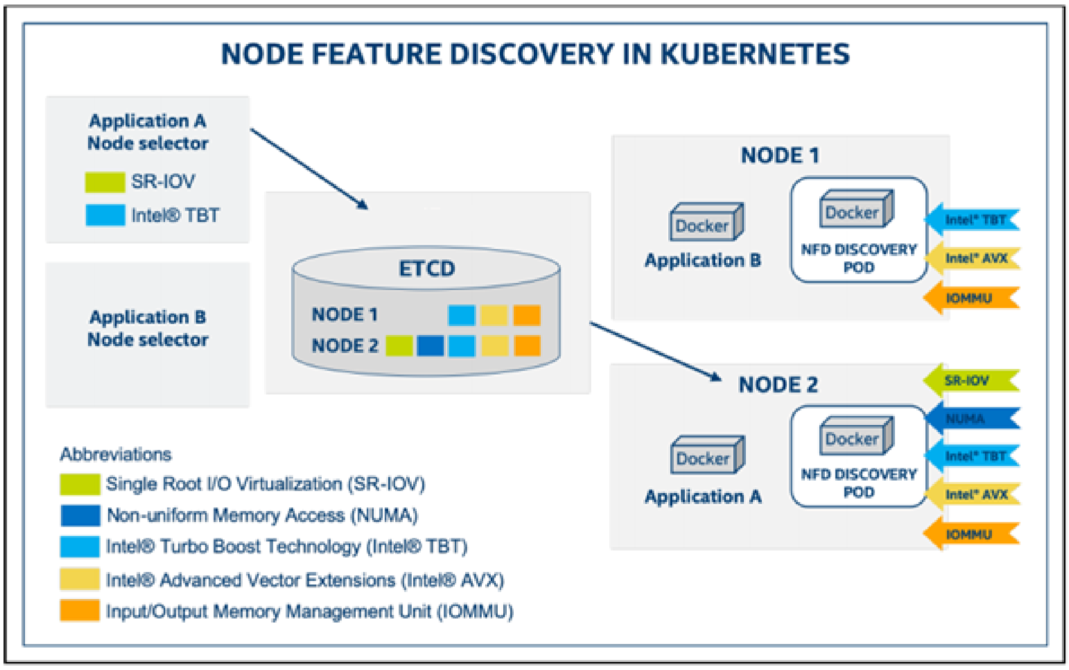


Figure 1. Node Feature Discovery in Kubernetes

NFD focuses on non-allocatable, qualitative features. It does not provide accounting or consumption capabilities. The allocatable resources from a worker node that require accounting, initialization, and other special handling are typically presented as Kubernetes Extended Resources and handled by device plugins. They are out of the scope of NFD.

The labels applied to the worker node object are aligned with the following format:

```
{
  "feature.node.kubernetes.io/cpu-<feature-name>": "true",
  "feature.node.kubernetes.io/custom-<feature-name>": "true",
  "feature.node.kubernetes.io/kernel-<feature name>": "<feature value>",
  "feature.node.kubernetes.io/memory-<feature-name>": "true",
  "feature.node.kubernetes.io/network-<feature-name>": "true",
  "feature.node.kubernetes.io/pci-<device label>.present": "true",
  "feature.node.kubernetes.io/storage-<feature-name>": "true",
  "feature.node.kubernetes.io/system-<feature name>": "<feature value>",
  "feature.node.kubernetes.io/usb-<device label>.present": "<feature value>",
  "feature.node.kubernetes.io/<file name>-<feature name>": "<feature value>"
}
```

A Kubernetes Pod may request specific features/capabilities in the infrastructure by populating the `nodeSelector` field in the Pod spec with a list of node feature labels. The example Pod spec snippet that follows shows how to request a worker node that is running on a server with Intel® Turbo Boost Technology and Intel® Advanced Encryption Standard New Instructions (Intel® AES-NI) enabled as well as with a Linux kernel version 5.15.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    env: test
  name: golang-test
spec:
  containers:
    - image: golang
      name: go1
  nodeSelector:
    feature.node.kubernetes.io/cpu-pstate.turbo: “true”
    feature.node.kubernetes.io/cpu-cpuid.AESNI: “true”
    feature.node.kubernetes.io/kernel-version.major": “5”
    feature.node.kubernetes.io/kernel-version.minor": “15”
```

NFD prevents the workload from starting if it does not find the requested features in the worker node. For Kubernetes clusters that have a single, consistent type of worker node allocated, all the worker nodes have the same functionality. In this scenario, NFD can be used to indicate if insufficiently capable worker nodes have been allocated to the cluster.

For heterogenous Kubernetes clusters, i.e., clusters with multiple types of worker nodes allocated, NFD takes on an equally impactful role by ensuring the different workloads being deployed land on the most appropriate types of worker nodes.

4.2 Platform Profile Composition

NFD is incredibly powerful in ensuring that the type of capabilities that the application developer requested for a deployment can be confirmed as available for deployment. However, there is a risk that overuse of NFD labels make it difficult to deploy workloads, particularly in heterogenous clusters. In essence, the more prescriptive the deployment spec is on the capabilities needed in the worker node, the more difficult it is to place that workload.

Platform Profile Composition (PPC) is a new usage model to help balance the power and flexibility of NFD with the needs to maximize workload placement potential and simplify the deployment scenario. PPC is a methodology based on the NFD `NodeFeatureRule` functionality.

The Kubernetes cluster administrator can create an NFD-specific Custom Resource with the `Kind NodeFeatureRule`. In this rule, the administrator defines the match rules and associated meta-label. If the rule is successfully matched on a worker node, NFD only applies the meta-label to the node object and does not label the node object with the dependent labels of the rule. Just as with NFD, these platform profiles are for qualitative properties only.

If the `NodeFeatureRule` is created under the default namespaces (`feature.node.kubernetes.io` or `<vendor>.feature.node.Kubernetes.io`, or `profile.node.Kubernetes.io` and sub-namespaces), there is also no requirement on the administrator to put the meta-label into the allow list.

Pod spec developers are only required to use the meta-label as the “platform profile” identifier.

An example of a `NodeFeatureRule` that the cluster administrator could create is:

```
apiVersion: nfd.kubernetes.test/v1alpha1
kind: NodeFeatureRule
metadata:
  name: example-nodefeaturerule1
spec:
  rules:
  - name: profile.node.kubernetes.io/example-nodefeaturerule1
    matchAll:
    - cpu.cpuid:
        "AESNI": { op: Exists }
        "AVX": { op: Exists }
        "AVX2": { op: Exists }
        "AVX512BW": { op: Exists }
        "AVX512CD": { op: Exists }
        "FMA3": { op: Exists }
        "VMX": { op: Exists }
    - cpu.pstate:
        "turbo": { op: IsTrue }
    - kernel.config:
        "NO_HZ": { op: IsTrue }
```

With this rule added to the Kubernetes cluster, the Pod spec could be simplified from having nine unique `nodeSelector` entries to just one entry as per the following example:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod-spec-leveraging-a-platform-profile
spec:
  containers:
  -  name: example-container
     image: docker.io/<OS>/tools:latest
     command:
     - /sbin/init
     resources:
       requests:
         cpu: "200m"
       limits:
         cpu: "200m"
  nodeSelector:
    profile.node.kubernetes.io/example-nodefeaturerule1: "true"
```

With methodical and proactive PPC rule definitions, the risk of both under and over specifying `nodeSelector` attributes in a Pod spec is reduced.

An area for future exploration with the Kubernetes NFD community is the option to define a set of PPC rules that are representative of the qualitative capabilities that are common to specific types of workload deployments. As these rules are defined with the minimal specification necessary for the use case, are published, and are automatically accessible in NFD-enabled Kubernetes distros, the consistency that PPC methodology offers will simplify the challenge of matching workloads to appropriate worker nodes.

5 Orchestrating for Compute Performance and Efficiency

For cloud and cloud-like deployments, the typical default perspective when requesting CPU, memory, and device resources is that they are consumed from a platform where the implicit ontology is of a flat structure devoid of any special hierarchy or distance relationships between resources. If you need 32 cores, 64 GB of memory, and one acceleration device and if the combination is available in the infrastructure, you are allocated these resources. However, this allocation model does not necessarily account for the processor silicon and server platform design where there is a hierarchy and distance relationship between components, potentially resulting in sub-optimal workload deployment.

Figure 2 depicts a dual socket 4th Gen Intel® Xeon® Scalable processor. In this example, Intel® Hyper-Threading Technology (Intel® HT Technology) is enabled so the operating system sees two sibling CPU cores per shared execution units with the example showing a total of 32 cores for the server[1].

Non-Uniform Memory Architecture (NUMA) means that there are different memory access behaviors depending on where the memory is accessed from. In Figure 2 there are four integrated memory controllers per socket with two DDR channels per controller depicted. The memory controllers can be configured in different modes including different Sub-NUMA Clustering (SNC) modes.

The I/O device affinity considerations are depicted in Figure 2 as there are two I/O devices shown per socket. The sockets are interconnected with an Intel® Ultra Path Interconnect (Intel® UPI) bus.

[1] Note, the number of CPU cores available is dependent on the specific CPU SKU that is used.

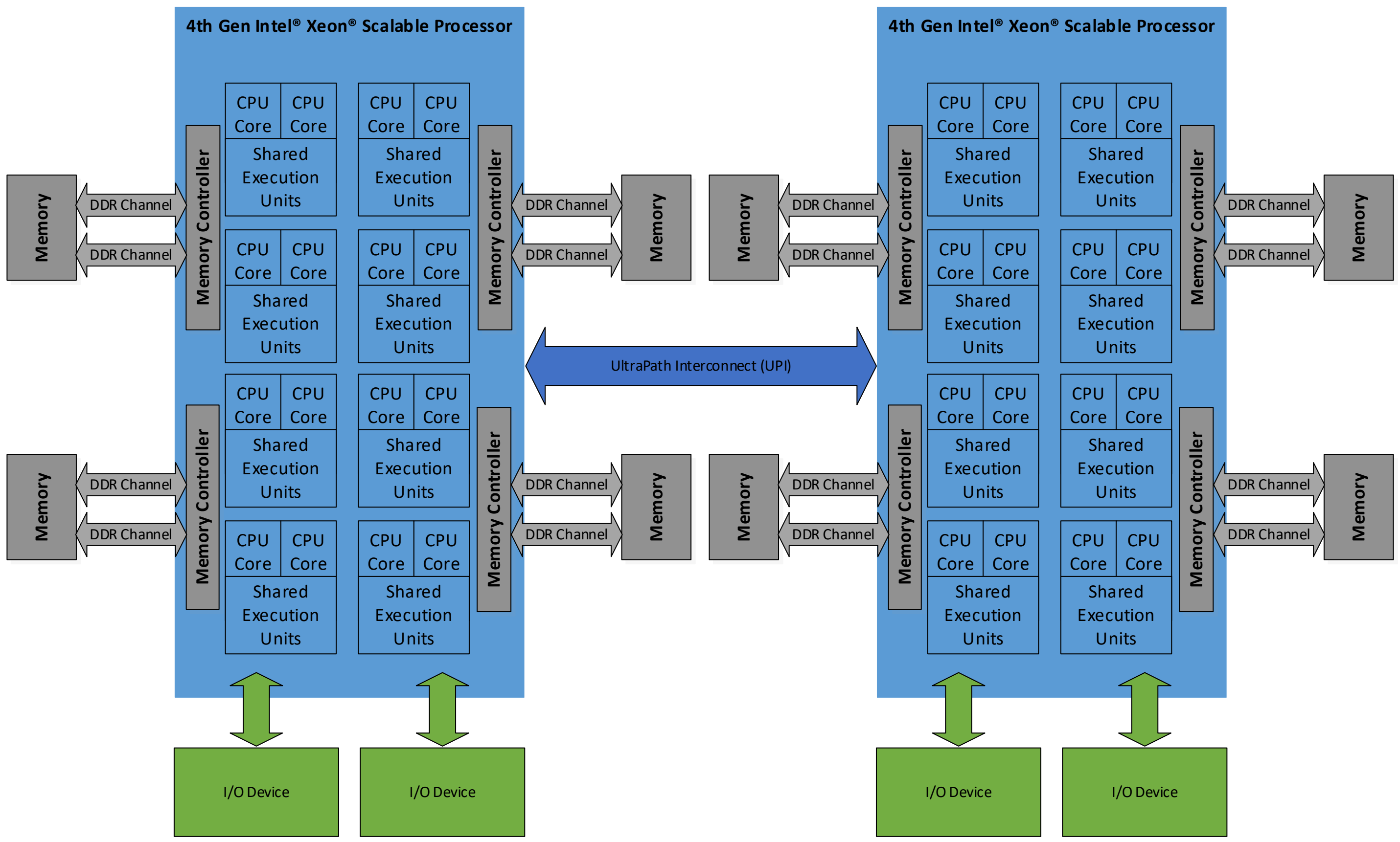


Figure 2. Graphic of Topology for the 4th Gen Intel Xeon Scalable Processor with Multiple Memory Controllers and I/O Devices

An optimal deployment of a workload would consider the physical topology of the system and where possible look to allocate CPU, memory, and I/O capacity from instances that have the best proximity to each other.

Different enablement approaches prevail on how to address the challenge of efficiently matching workloads to infrastructure with the desire that the workloads should be as agnostic to the infrastructure as possible. From a systems architecture perspective, there are kubelet-level options and container-runtime level options with distro and deployment considerations.

From a persona perspective, at one end of the spectrum is the view that the user should be able to refine the infrastructure to suit the workload as they are best positioned to understand what they need of infrastructure for their workload. At the other end of the spectrum is the view that the cluster administrator defines how the infrastructure behaves, is best placed to manage infrastructure characteristics for all, and that the workload must be fit within that definition. In practice, solutions tend to be designed with a specific systems architecture alignment and a persona model with a dominant perspective that can adjust to some degree over and back on the spectrum within the bounds of the guardrails defined by that solution.

Indicators on where a solution fits on this spectrum can be gleaned from understanding the persona responsible for setting resource allocation policy and the persona authorized to subsequently configure resource allocation policy.

When determining which model is best for a deployment understanding, the dynamicity of the resource allocation configuration is an important deciding factor. For example, some models require agent or kubelet reboot events that challenge the stability of other subsystems such as the network configuration.

5.1 Native Kubernetes CPU, NUMA, and Device Locality

In Kubernetes there is a set of managers operating at the kubelet level in the stack that work in collaboration to deliver a resource allocation model that is aware of physical server characteristics and account for CPU, NUMA, and device locality. Collectively they decide what resources to allocate to Pods and send the imperative requests of whatever container runtime layer has been deployed. The agnostic view of the container runtime choice is facilitated by Container Runtime Interface (CRI) specification.

5.1.1 Kubernetes Topology Manager

The Kubernetes Topology Manager [Ref 4] takes hints from the CPU Manager [Ref 5], Device Manager, and Memory Manager [Ref 6]. This set of capabilities moved to beta feature status in Kubernetes V1.18.

The Topology Manager policy can be set per worker node, with four options on policy `{none (default), best-effort, restricted, single-numa-node}`. The `none` policy does not apply any CPU topology consideration. The `best-effort` policy attempts to align Pods with NUMA zones but admits a Pod even if the alignment cannot be achieved. `Restricted` policy fails to admit a Pod if the NUMA alignment cannot be set, putting the Pod into a `terminated` state. Note, a Pod in a `terminated` state is not rescheduled by default, so another mechanism is required to trigger the scheduler, such as a `replicaset`, `daemonset`, or alternative mechanism. The `single-numa-node` is similar to the `restricted` policy but aims for deployment within a single NUMA zone.

5.1.2 Kubernetes CPU Manager

The CPU policy has two choices, `none` and `static`. The default `none` policy means that the OS scheduler is responsible for CPU management, with CPU limits managed solely by the Completely Fair Scheduler (CFS) quota. The CFS in Linux handles CPU core allocation for executing processes. It has dual goals of keeping the system interactive while also maximizing CPU utilization. For some demanding, high-performance workloads the behavior of CFS can negatively impact the workload and should be disabled. CFS also applies to `burstable` and `guaranteed` QoS models as CPU limits are specified.

The `static` policy enables the CPU manager to enforce more accurate configurations. It provides `hints` to the Topology Manager to help coordinate the locality of the resource allocation request.

In `static` mode shown in Figure 3, the available CPU cores in the system are grouped into shared and exclusive CPU pools. The CPU pool exclusivity relates to other Pods and not to system services. The `-reserved-cpus` option can be used to take out cores from the shared pools for use by the kubelet or other system services.

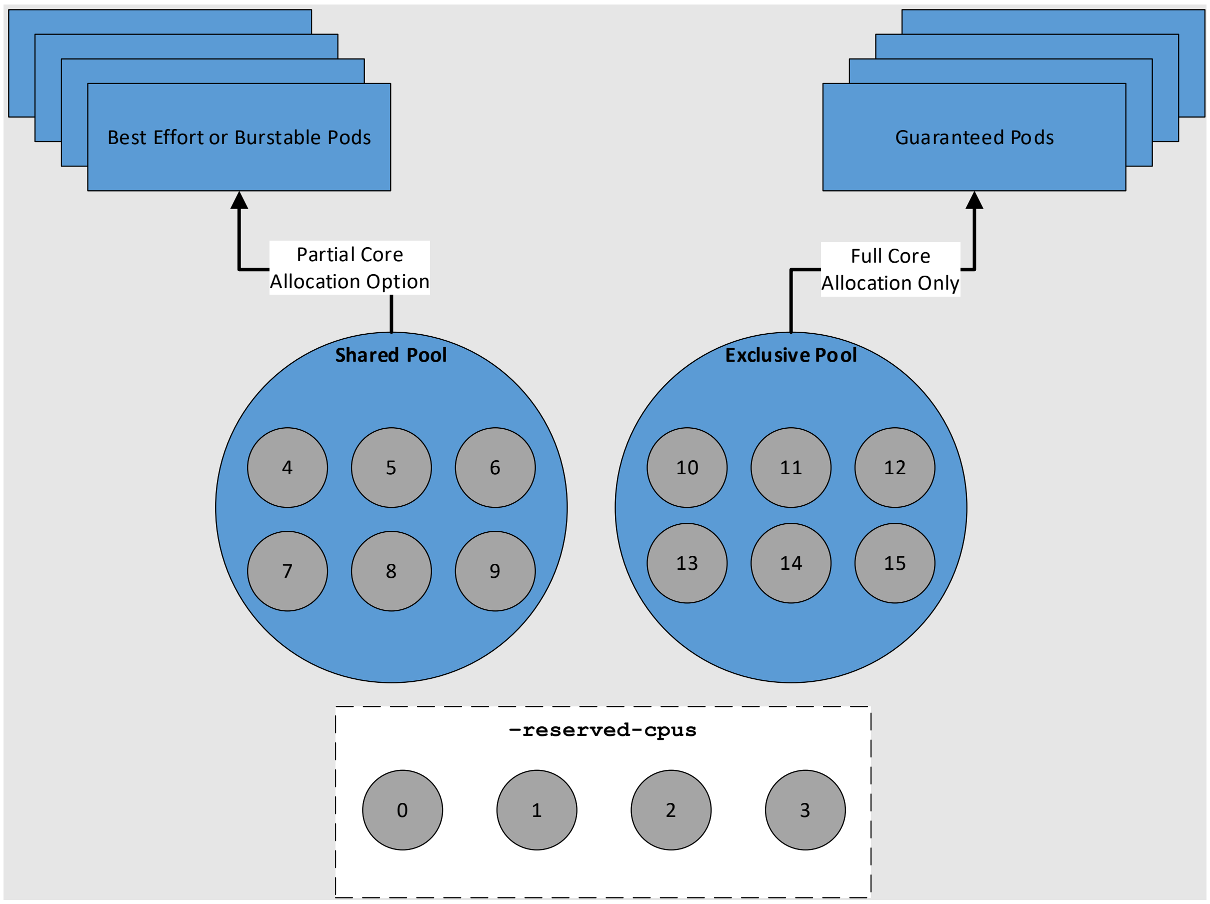

Best Effort or Burstable Pods
Guaranteed Pods
Partial Core Allocation Option
Full Core Allocation Only
Shared Pool
4
5
6
7
8
9
Exclusive Pool
10
11
12
13
14
15
-reserved-cpus
0
1
2
3

Figure 3. Kubernetes CPU Manager CPU Pool Grouping Model in `static` Mode

5.1.3 Kubernetes Device Manager

The Kubernetes Device Manager works with the Kubernetes Device Plugins to provide hints back to the Topology Manager to support selecting devices from the most appropriate locality.

The Kubernetes Device Plugin framework can work with the Kubernetes Topology Manager to support device affinity preferences where appropriate. The Kubernetes device plugins register with the Kubernetes device gRPC service. In doing so, they each advertise information about the device to the kubelet. The kubelet, in turn, shares this information with the API server.

The importance of these steps is to facilitate Kubernetes to support quantitative capacity allocation management of the devices.

Intel has enabled a set of Kubernetes device plugins [Ref 7] that support the Kubernetes Device Plugin framework. This set includes plugins for the following capabilities integrated with the 4th Gen Intel Xeon Scalable processor.

- Intel® QuickAssist Technology (Intel® QAT)
  - Intel QAT [Ref 8] is a symmetric key and public key cryptographic engine as well as a compression acceleration engine. It is intended for use in a wide variety of use cases such as edge and 5G performance, cost-effective storage, and powerful database analysis.
  - From a resource orchestration perspective, the unit of allocatable resources from an Intel QAT device to a container is represented by the PCIe Virtual Function (VF). The Kubernetes device plugin detects and advertises the Intel QAT VFs to the kubelet. When multiple Intel QAT PCIe Physical Functions (PFs) are available in the platform, the device plugin provides an allocation policy model set by the cluster administrator that chooses between balancing VF allocation between the PFs, or allocating all VFs from one PF before moving onto the next PF.
- Intel® Dynamic Load Balancer (Intel® DLB)
  - Intel DLB [Ref 9] is a hardware accelerator that manages a system of queues and arbiters connecting producers and consumers. It is intended for use in a variety of queue management and load balancing scenarios that have a significant CPU overhead in managing queues.
  - The unit of allocatable resources from Intel DLB is a PCIe Physical Function or Virtual Function under which a set of memory mappable regions (ports) and internal queue IDs are configured. The Kubernetes DLB device plugin detects and supports the allocation of these PFs/VFs to Pods.
- Intel® Data Streaming Accelerator (Intel® DSA)
  - Intel DSA [Ref 10] is a hardware accelerator for copying and transforming data, the latter inclusive of functionality for CRC checksum generation/verification, Data Integrity Fields, and memory page delta records.
  - The unit of allocatable resources from Intel DSA is a set of single-user dedicated work-queues and a set of multi-user shared work-queues that the DSA Kubernetes device plugin detects and supports the allocation of to Pods.
- Intel® In-Memory Analytics Accelerator (Intel® IAA)
  - Intel IAA is a compression and decompression hardware accelerator combined with primitive analytic functions particularly suited to database applications. The Intel® IAA plugin and operator optionally support provisioning of Intel IAA devices and work queues with the help of the accel-config utility through `initcontainer`.
  - The unit of allocatable resources from Intel IAA is a set of single-user dedicated work-queues and a set of multi-user shared work-queues that the IAA Kubernetes device plugin detects and supports the allocation of to Pods.
- Intel® Software Guard Extensions (Intel® SGX)
  - Intel SGX helps protect data in use via application isolation technology that protects selected code and data from modification using hardened enclaves.
  - The Intel SGX device plugin is responsible for discovering and reporting Intel SGX device nodes and capacity to the kubelet. The node-level configuration sets the limit on the number of enclaves in the environment and the number of containers that can share each enclave.
  - The device plugin grants the containers access to the enclave node and tracks the requested and allocated enclave memory to the containers. It also manages access to the "provision" capability that offers containers the additional capability to generate Intel SGX quotes supporting remote attestation workflows.

In addition to the device plugins for capabilities integrated in the 4th Gen Intel Xeon Scalable processor, Intel has created Kubernetes device plugins for resource management of the following peripherals.

- Kubernetes Intel GPU Device Plugin
  - The GPU device plugin supports logical GPU device modelling to address the device-per-Pod assumption within Kubernetes. Intel GPU memory and milli-core are modeled as extended resources and facilitate the GPU Aware Scheduling extender [section 7.2] to perform the node-level accounting function.
- Kubernetes Intel® FPGA Device Plugin
  - The Intel FPGA device plugin supports exposing information related to Intel® Arria® 10 devices and Intel® Stratix® 10 devices to the kubelet for FPGA allocation to Pods.
- Kubernetes Intel VPU Device Plugin

  - This device plugin supports exposing information to the kubelet related to the Intel VCAC-A card, the Intel Mustang V100 card, the Gen 3 Intel® Movidius™ VPU HDDL VE3 card, and the Intel® Movidius™ S VPU card.

An Intel Device Plugin Operator supports lifecycle management events for these plugins including installation, configuration, termination, and for some upgrade cycles.

5.1.4 Kubernetes Memory Manager

The Kubernetes Memory Manager is focused solely on the `guaranteed` QoS class of Pod. It supports reserving memory for `guaranteed` Pods as well as supporting `hugepage` allocations. As with other resource managers, it provides hints to the Topology Manager (which makes the decision) and then performs the reservation step.

The behavior of the Memory Manager is set with per-worker node policy options similar to the CPU Manager. The default `none` policy means that Memory Manager takes no action, while the `static` policy enables it to perform the memory reservation and `hugepage` allocation actions.

5.1.5 Challenges with Kubernetes Topology-Related Resource Managers

The functionality provided by the set of kubelet managers that work with the Kubernetes Topology Manager covers a broad section of common use cases and provides a significant performance and/or predictability improvement for many users over the innate simplistic flat model of resources. However, the architecture and functionality have several limitations that make supporting some more demanding use cases, such as those related to high performance networking, difficult or unnecessarily expensive/wasteful of precious resources.

Examples of limitations include:

- There is an inability to mix exclusive core allocation with shared core allocation to containers within a Pod. There is also an inability to allocate capacity to the Pod for sharing between one or more containers. Both limitations drive up Total Cost of Ownership (TCO) as lower intensity parts of the workload are forced into full core allocation even when they may only need a portion of a core.
- The policy configuration is spread out between multiple components with complex interactions, making it an intricate set of features to consume for all but the most advanced users.
- Adding new functionality typically requires touching multiple managers, which makes progressing the capabilities slow and cumbersome.
- Heterogenous CPU architectures, i.e., architectures with different types of compute cores in the same processor, are not supported.
- Topology affinity is statically tied to node level with a lack of flexibility to introduce new hardware capabilities such as are possible with Compute Express Link (CXL)-enabled technology [Ref 11].
- NUMA zone scaling is limited to eight NUMA zones.
- Insufficient control provided to users.

For these reasons, an update to the architecture of kubelet and managers is required to unleash faster innovation and support more demanding use cases. The Kubernetes community is aware of these topics. However, as the kubelet and managers are a core part of Kubernetes and have fundamental impact on stability and user experience, considerable care, due diligence, and prudence are applied to any related changes.

While the community and Intel collaborate on longer-term solutions, such as developing a plugin model for kubelet, Intel has developed the CPU Control Plane Manager and the CRI-Resource Manager to address the limitations.

5.2 Intel CPU Manager for Kubernetes (CMK) - Retired

Intel's initial investment in CPU management functionality for Kubernetes was known as the Intel CPU Manager for Kubernetes (CMK) [Ref 12]. This was an alternative to some of the Kubernetes native functionality noted in section 5.1.2. It was effective in demonstrating the value of advanced CPU management for the community and it addressed several specialist use cases required by some deployments.

Although CMK provided some unique value, on-going investment in CMK had several challenges such as:

- Functionality in the Linux kernel such as `isolcpus` that CMK depended on is effectively deprecated (although still in use) and the general guidance is to use `cpusets` instead. Upgrading to `cpusets` was going to require code rework.
- CMK was not compatible with the resource managers noted earlier.
- CMK only operated with Kubernetes `guaranteed` QoS Pod classes.
- CMK did not have a path to be adopted as part of an upstream project and hence was leading to bifurcation of focus on CPU management.

For these reasons, Intel ceased investment in this component.

5.3 Intel CPU Control Plane Manager for Kubernetes

The Intel CPU Control Plane Manager is a new kubelet-level CPU management offering for Kubernetes. It is designed from the outset to replace CMK, to add functionality such as namespace-related affinity models, to support the full set of Pod QoS classes inclusive of `best-effort`, `burstable`, and `guaranteed`, to have a path towards alignment with the proposed kubelet architecture refactor being discussed with the Kubernetes sig-node community, and to provide users the ability to set the allocation models for their workloads.

Figure 4 shows how the three Intel CPU Control Plane Manager components are deployed in a simplified cluster. They are packaged as three containers within a `daemonset`. The Intel CPU Control Plane Agent container is monitoring the Kubernetes API Server for Pod lifecycle events. The Intel CPU Control Plane daemon running on each worker node receives the Pod lifecycle events info from the agent and uses these triggers to apply CPU configuration via a `cpuset` to the Pods that the kubelet manages. The Intel CPU Control Plane Monitor container monitors the Pod resource allocations.

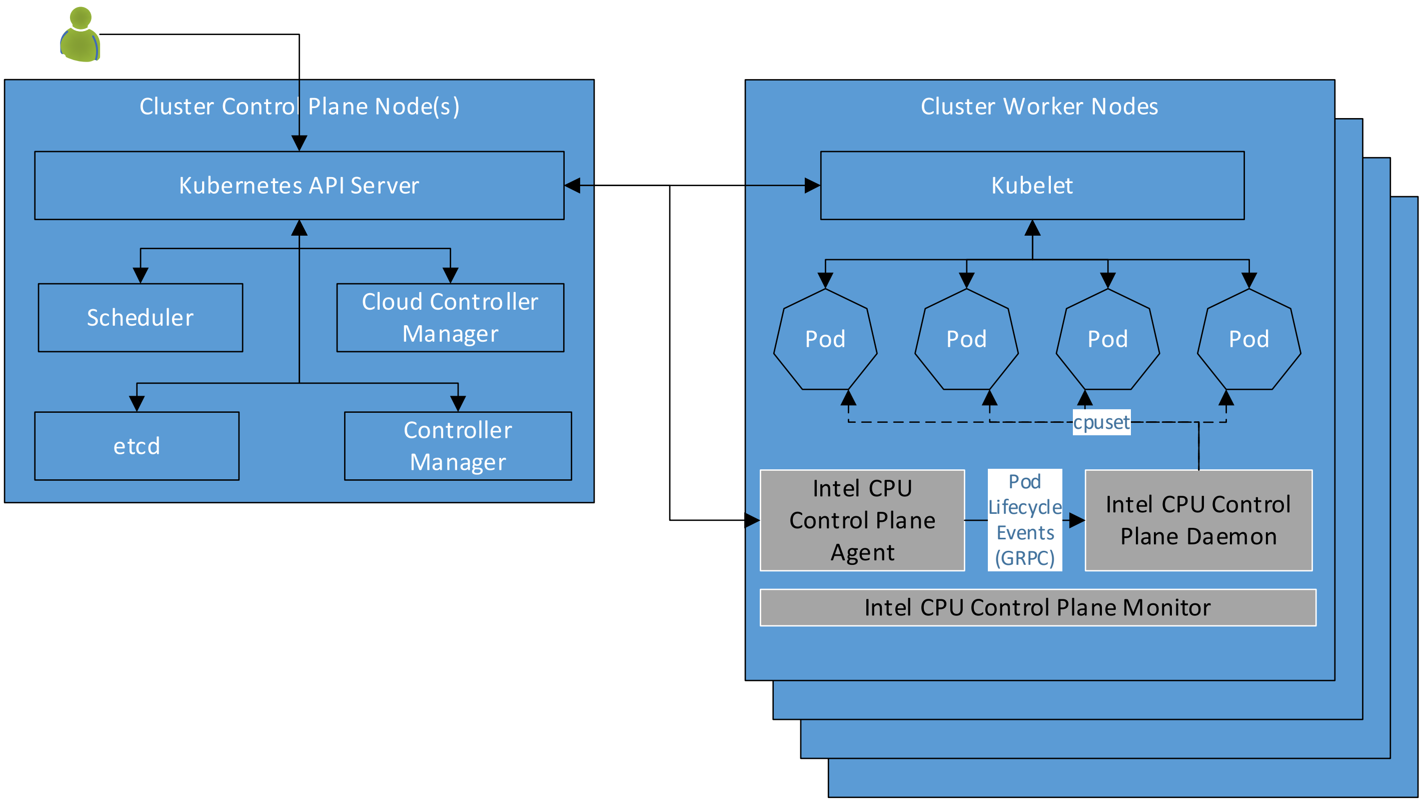

Cluster Control Plane Node(s)
Kubernetes API Server
Scheduler
Cloud Controller Manager
etcd
Controller Manager
Cluster Worker Nodes
Kubelet
Pod
Pod
Pod
Pod
cpuset
Intel CPU Control Plane Agent
Pod Lifecycle Events (GRPC)
Intel CPU Control Plane Daemon
Intel CPU Control Plane Monitor

Figure 4. Intel CPU Control Plane Manager for Kubernetes

The configuration that the Intel CPU Control Plane daemon uses to decide what CPUs to allocate is controlled by a set of allocation policies that can be modified by the cluster admin dynamically without the Intel CPU control plane or kubelet needing a reboot.

The policies have implemented an alternative, more nuanced interpretation of the default set of implicit Kubernetes QoS types. Specifically, `guaranteed`, `burstable`, and `best-effort` are parsed at a per-container level instead of only at a per-Pod level.

For some workloads, the default interpretation through the CPU manager `static` policy of the Kubernetes QoS types is overly restrictive and expensive. For example, if one container in a Pod requires exclusive access to cores (for a high-performance workload), the application developer would be required to specify that all containers within that Pod also need full core allocations, even if unnecessary. This leads to increased TCO and underutilized resources in the infrastructure.

The CPU Control Plane Manager has addressed this and other requirements through the implementation of the following policies that can be set by the user or by the cluster administrator.

- `default`
  - The default policy assigns each guaranteed container to an exclusive subset of CPU cores. CPU cores are taken sequentially from a list of available CPU cores.
  - `Guaranteed`, `burstable`, and `best-effort` QoS containers are not pinned to CPU cores with this policy.
- `numa`

- This policy assigns each `guaranteed` Pod container to an exclusive subset of CPU cores with minimal topology distance between the allocated CPU cores.
- `Burstable` and `best-effort` containers within the Pod are not pinned.

- `numa-namespace`
  - This policy isolates each namespace to a separate NUMA zone. The system must support enough NUMA zones to assign separate zones to each namespace.
  - `Guaranteed` containers CPUs are shared with `burstable` and `best-effort` containers, but not with other `guaranteed` containers.
- `numa-namespace-exclusive`
  - This policy is similar to `numa-namespace`, except that it assigns exclusive access to CPU cores to `guaranteed` QoS containers with no sharing with `burstable` or `best-effort` containers within the Pod.

5.4 Intel Container Runtime Interface – Resource Manager (CRI-RM)

The Intel Container Runtime Interface - Resource Manager (CRI-RM) is a hardware-aware container runtime extension [Ref 13]. It provides a flexible way to try different resource management algorithms, known as policies. The purpose of the policies is to allow the cluster administrator to express various resource allocation preferences and strategies related to addressing NUMA affinity, noisy neighbor, and dynamic power management challenges.

There are several mutually exclusive policies that the cluster admin can set to control how CRI-RM allocates resources to Pods on a per-worker node basis or to each member of a group of worker nodes. The set of policies in CRI-RM range from hardware-centric policies with topology-aware memory tiering support to more application-centric policies such as the PodPools policy and Balloons policy. Policies may be changed by the cluster admin by adhering to a CRI-RM reset sequence without requiring a kubelet reset. Configuration within each policy may be changed dynamically by the cluster administrator with a ConfigMap that exists in the kube-system namespace processed by the CRI-RM Resource Manager Agent.

5.4.1.1 CRI-RM Deployment Models

CRI-RM can be leveraged with the Kubernetes upstream stack in two different deployment models.

- Container Runtime Proxy Mode
  The cluster administrator inserts CRI-RM in the architecture between the kubelet and container runtimes such as `containerd` and `CRI-O`. When CRI-RM is first deployed in proxy mode, the kubelet requires a restart to point it at the CRI-RM socket. During this kubelet restart, it is recommended to disable kubelet-level resource managers as CRI-RM takes over the handling of Kubernetes native resources, including CPU and memory. In addition to native resources, CRI-RM includes support for a suite of Intel technologies related to power management, memory tiering, and cache and memory controls.
- NRI Plugin Mode
  CRI-RM is transitioning from being solely a CRI proxy in the architecture to being a plugin into the container runtime. This new container-runtime plugin interface is the Node Resource Interface (NRI) [Ref 14]. After the NRI work is complete in the target containerd [Ref 15] and CRI-O [Ref 16] runtime communities, CRI-RM will also be deployable with an NRI interface model. The main functionality of CRI-RM is not expected to change during this transition. However, the packaging of policies may be split into individual NRI plugins (subject to community agreement). In NRI mode, a kubelet restart is not required; however, it continues to be appropriate that the kubelet-level managers be disabled in this configuration.

5.4.2 Topology Aware Policy

The Topology Aware policy [Ref 17] builds an internal model of the worker node in a four-layer tree structure.

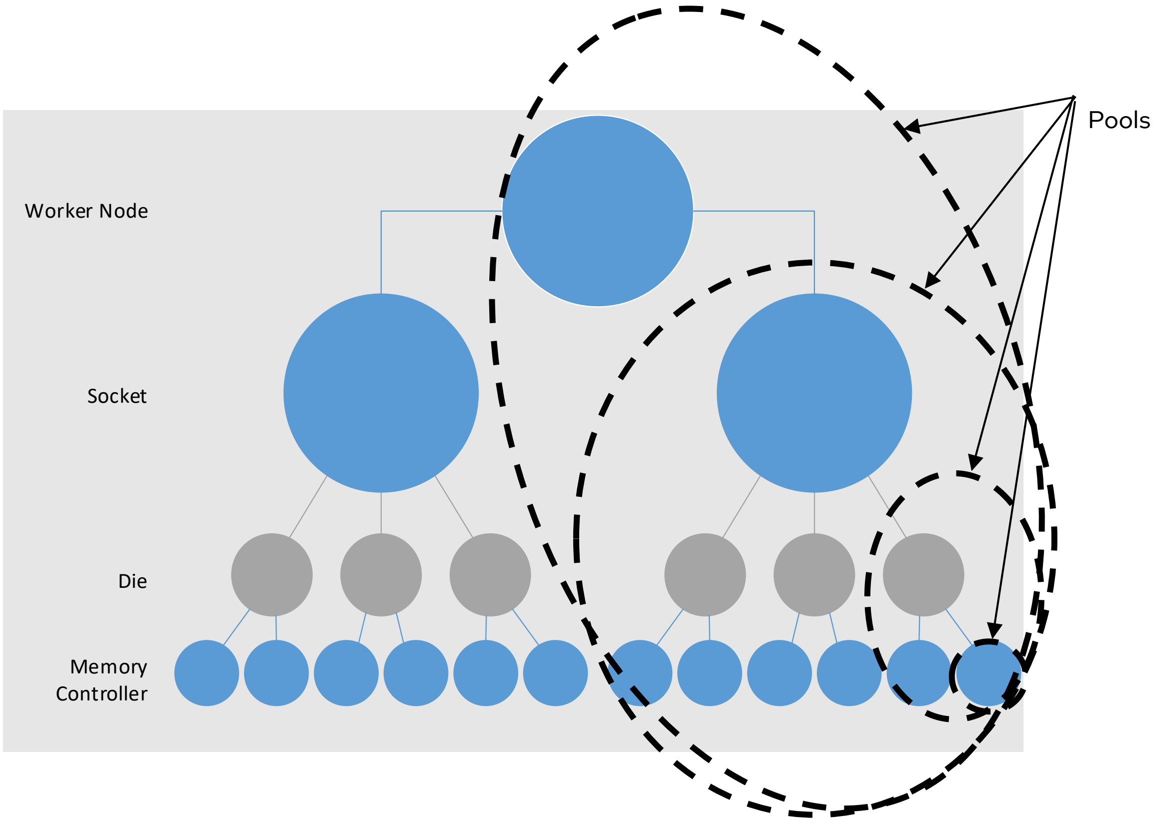


Figure 5. Kubernetes CRI-RM Topology Aware Policy Model

At the lowest layer in the tree hierarchy, the "leaf" nodes represent the memory controllers with their associated memory and the CPU cores that are closest to this memory in terms of access latency.

The concept of pools is overlaid on this with pools higher up the tree containing the resources of all subordinate pools. Pools at the same level in the hierarchy are populated with resources that have an innate separation in the architecture.

Resource allocation with this policy aims to fit workloads (i.e., Pods) into lower-level pools, pools that are least allocated, and pools with better alignment to devices such as accelerators.

The cluster administrator may further refine the operation of this policy with several configuration options such as `PinCPU`, `PinMemory`, `PreferIsolatedCPUs`, `PreferSharedCPUs`, `ReservedPoolNamespaces`, `ColocatePods`, and `ColocateNamespaces`.

Containers are automatically categorized into groups based on their QoS class and resource requirements. These groups (`kube-system` group, `low-priority` group, `sub-core` group, `mixed` group, and `multi-core` group) further refine the specific CPU core choice the CRI-RM CPU allocator chooses for the workload.

At a Pod and container scope, a user may choose to further refine the behavior related to shared, exclusive, and isolated CPUs by annotating the Pod spec.

The Topology Aware Policy also includes some explicit functionality related to Persistent Memory (PMEM) capabilities as enabled by Intel® Optane™ for workloads that have both DRAM and PMEM memory allocations. This includes a "Cold Start" capability that allocates DRAM after a timeout period so that allocated large unused memory does not need to be migrated to PMEM, and a Dynamic Page Demotion capability that actively moves rarely used pages from DRAM to PMEM.

5.4.3 Static-Pools Policy (STP)

The Static-Pools policy (STP) [Ref 18] implements a similar algorithm to the now-retired CPU Manager for Kubernetes (CMK) [Ref 20] functionality related to `cmk isolate`. This policy supports the same environment variables and configurations as CMK and hence offers a path for users of CMK to move to an actively developed project. Nonetheless, it is recommended that users switch from STP to one of the other policy models.

5.4.4 Balloons Policy

The Balloons policy [Ref 21] introduces an alternative CPU and memory grouping model, where CPU cores and optionally memory, are associated with the logical concept of a "balloon" that may grow or shrink. A set of policy parameters define how balloon-based allocation operates and a set of parameters that further describe the operation on a `BalloonTypes` basis.

The usage model intended for leveraging the Balloons policy is one that provides the cluster administrator the ability to restrict the OS scheduler preference for spreading workloads across all cores within a `cgroup`. By restricting how the applications are spread on the worker node, there is a greater potential for other OS functions such as the CPU Frequency Governor to support a power reduction policy through P-state and C-state configurations [section 6.1] without additional host OS administrative inputs.

The overall Balloon policy determines behavior such as limiting containers, i.e., "pinning" the containers to CPUs from the selected Balloon, and if the memory the container can leverage is from the same NUMA node as the CPUs allocated to the balloon. There is a default Balloon type, and the cluster administrator can define one or more `BalloonTypes` (Figure 6). The `BalloonTypes` setting determines the namespace that may leverage the Balloon as well as settings such as the minimum and maximum number of CPUs within the `Balloon`, if containers are restricted to a specific `Balloon` or may also leverage capacity from `Balloons` of the same type, what type of `CpuClass` is allocated to the `Balloon`. The `CpuClass` defines the core and uncore minimum and maximum frequency settings that are applied to all the CPUs as they are allocated to an instance of the `BalloonType`.

Within the Balloon policy, there is the concept of a class of "idle" CPUs, the `IdleCPUClass`. CPUs in this class do not belong to any of the defined Balloons. The idle class of CPUs can be configured to prioritize a power saving scheme where these CPUs are configured into lower power consumption states or alternatively configured to support high performance bursting of workloads within balloons.

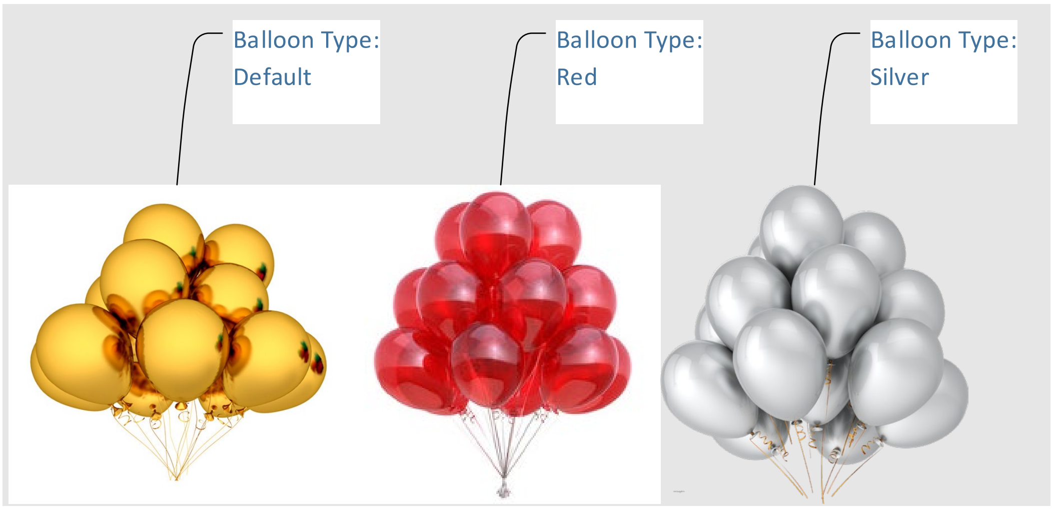


Figure 6. Kubernetes CRI-RM Balloon Policy Model

The user is responsible for determining the name and intention of the balloons and to make the balloon request within the Pod spec.

5.4.5 Podpools Policy

The Podpools policy model [Ref 22] supports the concept of Pods, inclusive of the aggregate of container-based CPU requests, fitting into the core capacity allocated to the `Podpool`. Each `Podpool` is configured by the administrator to have a number of properties such as the specific number of cores allocated to the `Podpool`, the maximum number of Pods that can be placed in each `Podpool`, and the percentage of non-reserved cores on the node that may be allocated to this type of `Podpool`. The latter setting implicitly defines the number of `Podpools` of the particular type.

The user may specify the `Podpool` name as an annotation in the Pod spec. If the user does not specify one of the `Podpools` in the system, then the Pod is deployed on the shared CPU cores that have not been allocated to any of the `Podpools`.

5.4.6 Container Affinity and Anti-Affinity Policy

The Container Affinity and Anti–Affinity policy model [Ref 23] provides a mechanism to control how near or far containers are placed on the node from each other. The distance between containers is a function of the server architecture, e.g., dual-socket servers with NUMA considerations, and processor architecture, e.g., Sub-NUMA Clustering (SNC) configuration.

The distance, aka affinity, relationship between the containers is captured as the scope of containers that the distance refers to, which containers are of relevance, and how intensely the policy algorithm pursues the affinity/anti-affinity requirement.

5.4.7 Block I/O QoS Class Resource

The Block I/O QoS class resource provides a mechanism to control block device I/O scheduling priority (weight), throttle the Block I/O bandwidth, and throttle the number of I/O operations.

CRI-RM applies block I/O controller parameters to Pods via the Linux cgroups Block IO controller (`blkio`).

5.4.8 CPU Allocator Component

Underpinning the CRI-RM policies is a CPU Allocator component. This component groups CPUs in `high`, `normal`, and `low` priority. Categorizing CPUs in the system into these priorities is dealt with in one of two methods.

- Intel® Speed Select Technology (Intel® SST)-based CPU prioritization
  In this mode, Intel® Speed Select Technology - Turbo Frequency (Intel® SST-TF) and Intel® Speed Select Technology - Base Frequency (Intel® SST-BF) high priority cores are added to the high priority class. For Intel® Speed Select Technology - Core Power (Intel® SST-CP)-enabled cores, the Class of Service (CLOS) settings determine what grouping the cores are added to.
- Linux `CPUFreq` prioritization
  CPU cores are divided into classes based on their Base Frequency and their Energy-Performance Preference (EPP) setting.

The CPU allocator tries to optimize the allocation of CPUs in terms of the hardware topology. More specifically, it aims at packing all CPUs of one request "near" each other to minimize memory latencies between CPUs.

5.5 Intel® Resource Director Technology (Intel® RDT) Kubernetes Enabling

Intel Resource Director Technology (Intel RDT) is a suite of capabilities for monitoring and controlling CPU cache and memory resources that are shared between concurrent users of the platform. By default, cache and memory resources are shared dynamically and automatically between concurrent users, giving a good level of performance. However, there are deployment scenarios where workloads benefit from a more precise allocation of these resources for greater workload predictability and performance. Intel RDT is composed of the following capabilities.

- Cache Monitoring Technology (CMT) can monitor the last-level cache utilization by individual threads and applications. CMT can be leveraged to improve workload characterization. It enables advanced resource-aware scheduling decisions and can aid noisy neighbor detection.
- Cache Allocation Technology (CAT) provides the capability to have application threads request a Cache Class of Service (CLOS); each CLOS is mapped to a configured portion of the available cache. CAT may be used to enhance runtime determinism and prioritize important applications through reduced cache contention.
- Code and Data Prioritization (CDP) is a specialized extension of CAT. CDP enables separate control over code and data placement in the second and third level caches. Certain specialized types of workloads may benefit with increased runtime determinism, enabling greater predictability in application performance.
- Memory Bandwidth Monitoring (MBM) enables multiple application threads to track their memory bandwidth. This technology further improves the detection of noisy neighbors, supports workload characterization and debugging of performance for bandwidth-sensitive applications, and contributes to the potential for more effective Non-Uniform Memory Access (NUMA)-aware scheduling.
- Memory Bandwidth Allocation (MBA) enables approximate and indirect control over memory bandwidth available to workloads, enabling new levels of interference mitigation and bandwidth shaping for noisy neighbors present on the system.

5.5.1 Linux Resource Control

Intel RDT is enabled in the Linux kernel with the `Resource Control` functionality. This exposes Intel RDT functionality to user space applications through the `resctrl` filesystem. `resctrl` structures resources into groups and resource groups are represented as directories in the `resctrl` filesystem. For more information on `resctrl`, see the latest Linux kernel documentation in [Ref 24] and [Ref 25].

5.5.2 Kubernetes Intel RDT Resource Allocation Configurations (containerd/CRI-O)

Intel RDT is enabled through two distinct paths for container deployments through Kubernetes. The first path is via extensions integrated into two container runtime projects, containerd (v1.6) and CRI-O (targeting release in v1.22). The second path is via an Intel Kubernetes add-on, Container Runtime Interface Resource Manager (CRI-RM). Both implementations present equivalent functionality.

The Intel RDT enablement in containerd, CRI-O, and CRI-RM depends on the Intel `goresctrl` library (https://github.com/intel/goresctrl). This library provides a Go interface to integrate with the native Linux user space `resctrl` filesystem. `goresctrl` provides a two-level hierarchical representation of RDT resources consisting of partitions and classes.

`goresctrl` partitions consist of available resources and classes that share the resources. Resources include portions of caches (L2 and L3) and memory bandwidth (MB). Cache partitioning is exclusive with no overlapping regions permitted while memory bandwidth is shared. `goresctrl` classes represent the RDT classes to which processes are assigned. Cache allocation between classes under a specific partition may overlap. A full description of class-based configurations is in the `goresctrl` manuals [Ref 26].

The Kubernetes administrator must set up the Intel RDT configurations for each node in the cluster. These settings may be changed at runtime by the Kubernetes administrator without requiring node agent resets/reboots. The Kubernetes administrator may choose a configuration that allocates cache and memory capacity based on the implicit Kubernetes Pod QoS class (`BestEffort`, `Burstable`, or `Guaranteed`) or create a configuration aligned with some alternative naming of class, such as bronze, silver, or gold.

An example of a Kubernetes QoS class configuration could be to allocate 60% of the L3 cache lines for one partition exclusively for use by `Guaranteed` QoS classes of Pods. The remaining 40% of L3 cache could be allocated to another partition for sharing between `burstable` and `BestEffort` QoS classes of Pods, with `BestEffort` Pods limited to only 50% of this. `Guaranteed` QoS class Pods could get full memory bandwidth whereas the other Pod classes could be throttled to 50%.

Containerd configuration information is available in the published manuals [Ref 27] and the following is an extract of the containerd `rdt_config_file` described in the `goresctrl` manual [Ref 28].

```
Options:
  l2:
    optional: true
  l3:
    optional: true
  mb:
    optional: true
partitions:
  exclusive:
    # Allocate 80% of all L2 cache IDs to the “exclusive” partition
    l2Allocation: “80%”
    # Allocate 60% of all L3 cache IDs to the “exclusive” partition
    l3Allocation: “60%”
    mbAllocation: [“100%”]
    classes:
      guaranteed:
        # Allocate all of the partitions cache lines and memory bandwidth to “guaranteed”
        l2Allocation: “100%”
        l3Allocation: “100%”
        # The class will get 100% by default
        #mbAllocation: [“100%”]
  shared:
    # Allocate 20% of L2 and 40% L3 cache IDs to the “shared” partition
    # These will NOT overlap with the cache lines allocated for “exclusive” partition
    l2Allocation: “20%”
    l3Allocation: “40%”
    mbAllocation: [“50%”]
    classes:
      burstable:
        # Allow “burstable” to use all cache lines of the “shared” partition
        l2Allocation: “100%”
        l3Allocation: “100%”
        # The class will get 100% by default
        #mbAllocation: [“100%”]
      BestEffort:
        # Allow “BestEffort” to use all L2 but only half of the L3 cache
        # lines of the “shared” partition.
        # These will overlap with those used by “burstable”
        l2Allocation: “100%”
        l3Allocation: “50%”
        # The class will get 100% by default
        #mbAllocation: [“100%”]
      DEFAULT:
        # Also configure the resctrl root that all processes in the system are
        # placed in by default
        l2Allocation: “50%”
        l3Allocation: “30%”
        # The class will get 100% by default
        #mbAllocation: [“100%”]
```

When the Kubernetes administrator configures Intel RDT to align with the Kubernetes QoS classes, the Pod spec creator does not need to add any Intel RDT configuration to the Pod spec. The implicit Pod spec QoS class is used to select the correct Intel RDT configuration for that Pod. This model offers a simplicity in consumption but has the implication that the Kubernetes administrator is setting a custom behavior within the cluster that the application developer may not be aware of.

If the Kubernetes administrator configures Intel RDT with a custom set of classes (e.g., `bronze`, `silver`, `gold`), the Pod spec author must add annotations to the Pod spec to activate the Intel RDT configuration. The annotations can apply at the Pod-level or on individual containers within the Pod, where the per-container annotation takes priority over any related Pod-level annotation that may exist. The example that follows sets container 1 and 3 to an RDT `bronze` configuration and sets container 2 to an RDT `gold` configuration.

```
apiVersion: v1
kind: Pod
metadata:
  name: test
  annotations:
    rdt.resources.beta.kubernetes.io/pod: bronze
    rdt.resources.beta.kubernetes.io/container.container2: gold
spec:
  containers:
  - name: container1
    image: k8s.gcr.io/pause
  - name: container2
    image: k8s.gcr.io/pause
  - name: container3
    image: k8s.gcr.io/pause
```

The approach requires a modification to the Pod spec but has the benefit that it is more likely the application developer will understand the implications of the Intel RDT configuration that will be activated for their workload.

If the Kubernetes administrator has chosen to deploy CRI-RM in the cluster, then there is no dependency on containerd or CRI-O for this support. With CRI-RM, the Intel RDT behavior as outlined earlier is also available and may be combined with additional advanced resource management behaviors. For complete documentation on how Intel RDT can be used with CRI-RM, refer to the Intel RDT portion of the CRI-RM documentation [Ref 29].

At the time of writing, an enhancement to Kubernetes is being implemented (KEP-3008) that improves the support in Kubernetes (via CRI updates) for class-based resource handling such as is applicable to Intel RDT and Block IO. The initial primary interface continues to be via Pod annotations as outlined in this section.

5.5.3 Kubernetes Intel Resource Management Daemon (RMD) - Retired

The Intel Resource Management Daemon (RMD) was built to simplify the use of Intel RDT and to then develop into a central interface for hardware resource management on Intel platforms. The initial implementation focused on Intel RDT Cache Allocation Technology enabling. The simplified interface provided a mechanism to partition the cache into separate groups with fixed and burstable capacity.

Since the RMD project inception, RDT CAT enabling is now available natively in two container runtimes, containerd and CRI-O. The usage model enabled through Kubernetes now allows for containers on bare metal to easily benefit from CAT based on the cluster administrator configuration. The initial goal of the RMD project has been addressed while simplifying the deployment configuration.

As a result, Intel made the decision to retire this project and the related Kubernetes RMD operator.

6 Orchestrating for Power

Cloud native application developers face several challenges in balancing key performance indicators for their application, such as low jitter/latency or high performance/throughput, and balancing that with the ever-growing market demand to have a sustainable product. To that end, orchestrating for power is a requirement that is growing in importance.

6.1 Intel Power Management Capabilities

Intel Xeon Scalable processors have a rich feature set of advanced power management capabilities that were designed to allow users to adjust the compute performance to power consumption ratio dynamically in response to the needs of the workload or the infrastructure owner.

Intel Speed Select Technology (Intel SST) is the collection technologies giving users granular control over many aspects of the CPU power such as:

- Intel Speed Select Technology - Base Frequency (Intel SST-BF)
  The base frequency is the maximum, non-turbo, frequency of the core. This feature provides the ability to raise the base frequency of some cores and reduce that of other cores. This is particularly useful for workloads where some of the processes are CPU bound and could benefit from the increased CPU frequency.
- Intel Speed Select Technology - Core Power (Intel SST-CP)
  This feature provides the ability to inform the power management unit of a priority identifier, a Power QoS, so that it can allocate power headroom in a priority or proportional order. This functionality works with other Intel SST features.
- Intel Speed Select Technology - Turbo Frequency (Intel SST-TF)
  This feature provides the ability to further increase all or some of the core frequencies beyond the base frequency, based on factors within the processor such as temperature, power consumption, and load.

The Intel Power Management Technology Overview provides a detailed description of these capabilities [Ref 30].

6.1.1 Core Sleep/Idle States (C-state)

The Advanced Configuration and Power Interface (ACPI) specification [Ref 31] defines the processor power state, known as its C-state. The C-state is sometimes colloquially known as the processor "idle" state on a per-core basis or on a CPU package basis. Core and package C-states coordination is managed by the CPU Power Control Unit. C-state values range from C0 to Cn, where n is dependent on the specific processor. When the core is active and executing instructions, it is in the C0 state. Higher C-states indicate how deep the CPU idle state is.

Higher C-states consume less energy when resident in that state but require longer latency times to transition into the active C0 state.

The BIOS/UEFI configuration can be configured to restrict how deeply the cores and package can idle, e.g., it is possible to restrict access to the deepest C-state.

In Linux, C-state management is implemented on modern Intel Xeon processors with the `intel_idle` driver that is part of the CPU idle time management subsystem. Linux categorizes a CPU core as being idle if there are no tasks to run on it except for the "`idle`" task.

6.1.2 Core Frequency/Voltage State (P-state)

Intel Xeon Scalable processors include the ability to alter the processor operating frequency and voltage between high and low levels. The frequency and voltage pairings are defined in the ACPI specifications as the processor Performance State (P-state). P-states are SKU-specific settings ranging from the low end with minimums defined by `Pn` to the Base Frequency (`P1`), modifiable by Intel SST-BF, to the maximum single core turbo (`P01`).

Intel Xeon Scalable processors can manage P-state transitions with HW P-state Management (HWP), or the Linux operating system can request P-states via the `intel_pstate` driver. The controls are exposed to host OS user/administrators with Linux `sysfs` filesystem and utilities such as the `CPUFreq` Governor [Ref 32].

In some deployments, the host OS administrator chooses to configure P-state policies via the `CPUFreq` Governor to the node aligned with desired behaviors, such as maximizing performance (`performance` policy) or minimizing power consumption (`powersave` policy). Other deployment models can benefit from more targeted management of P-states.

6.1.3 Uncore Frequency Scaling (UFS)

Intel describes functional units of the processor outside of the CPU core but closely connected to it as the uncore. Functions such as cache or memory controllers are part of the uncore. The uncore functional blocks in Intel Xeon Scalable processors also include an Uncore Frequency Scaling (UFS) capability to scale their frequency in support of performance or power saving behaviors. The default setting is for the uncore frequency to be scaled dynamically by the processor based on load and power state.

UFS is supported in Linux and exposed for configuration via the `sysfs` filesystem [Ref 34]. This offers the ability to query the discovered minimum, active, and maximum uncore frequency settings as well as provides the ability to set min and maximum uncore frequencies.

6.2 Power Manager for Kubernetes* software

The Power Manager for Kubernetes* software [Ref 34] is a Kubernetes Operator that has been developed to provide cluster users with a mechanism to dynamically request adjustment of worker node power management settings applied to cores allocated to the Pods. The power management-related settings can be applied to individual cores or to groups of cores, and each may have different policies applied. It is not required that every core in the system be explicitly managed by this Kubernetes power manager. When the Power Manager is used to specify core power related policies, it overrides the default settings.

The container deployment model in scope is for containers running on bare metal (i.e., host OS) environments.

The Power Manager for Kubernetes software has two main components, the Configuration Controller and the Power Node Agent, which in turn has a dependency on the Intel® Power Optimization Library.

The Configuration Controller deploys, sets, and maintains the configuration of the Power Node Agent. By default, it applies four cluster administrator or user modifiable `PowerProfile` settings to the Power Node Agent. This facilitates Pods to choose between a `performance`, `balance-performance`, `balance-power` profile, and a `default` profile, to be configured on cores allocated to the Pod.

The Power Manager automatically detects the minimum and maximum CPU frequency limits on the worker node and dynamically populates the profile-specific CPU frequency attributes within these ranges. The ratios chosen for calculating these frequency values are based on the `PowerProfile` target Intel SST-CP Energy Performance Preference (EPP) settings. The EPP setting also configures the Power QoS configuration when used with Intel SST-CP, which in turn directs the hardware to allocate power headroom in a priority or proportional order between cores.

The default profile is configured to match with the `power` profile, which defaults to the most aggressive CPU power saving configuration and is modifiable. This configuration is applied to the shared pool.

The Power Node Agent is a Kubernetes `DaemonSet` that is deployed to targeted worker nodes in the cluster. It monitors Pod lifecycle events and is responsible for activating the requested `PowerProfile` configuration to cores allocated to the Pods.

The Intel Power Optimization Library [Ref 35] is an independently consumable golang library where the data model for the power management configuration is defined. Logically, cores are grouped into:

- A shared pool
  - The Power Manager for Kubernetes software uses this to represent the Kubernetes shared CPU pool.
  - Within the shared pool, the subset of system-reserved cores is grouped under a default pool. Cores in the default pool are not subject to power setting configurations.
- An exclusive pool
  - The Power Manager for Kubernetes software uses this to group cores allocated to `Guaranteed` Pods. When cores are allocated to `Guaranteed` Pods, they are moved from the Shared Pool into the Exclusive Pool.

Traditionally C-state configuration is set by a server administrator and is considered a relatively static configuration. For workloads that cannot tolerate the transition latency from a deep C-state to the active C0, it has been necessary to restrict the entire server from entering the deeper C-states. For vertically integrated appliances, finer grained C-state control was achievable due to the fully integrated nature of the appliance. However, for cloud native deployment models that was not the case.

The Power Manager for Kubernetes software provides the ability for a cluster administrator or user to dynamically configure the permitted C-state levels applied to logical groupings of cores split between a shared pool for `burstable` and `BestEffort` Pods, an exclusive pool for `Guaranteed` Pods, or an individually specified set of cores.

6.3 Intel CRI-RM Power Management

For information on how Intel CRI-RM has implemented power related configurations, refer to section 5.4.

7 Orchestrating Based on Observability

A rich set of infrastructure data is readily accessible in a cloud native environment via telemetry. Intel Platform Aware Scheduling [Ref 36] provides a set of projects that are designed to leverage infrastructure telemetry to influence scheduling decisions. Intel Platform Aware Scheduling includes Telemetry Aware Scheduling and GPU Aware Scheduling, described in the next sections. Both technologies can be used simultaneously.

7.1 Telemetry Aware Scheduling

Telemetry Aware Scheduling (TAS) is a generic framework that adds a control loop to Kubernetes scheduling that supports policy-based actions to be performed on the Kubernetes Pods based on a specified set of telemetry data.

TAS is implemented as a Kubernetes scheduler extension. It has a generic capability to look at one or more metrics via the Kubernetes Custom Metrics API, compare the metrics to a user specified value, and invoke an action based on the result. The Custom Metrics API is fulfilled by a custom metrics pipeline. The custom metrics pipeline is a Kubernetes distribution or cluster specific installation.

The Intel reference deployment model uses Prometheus as the Time-Series Database, the Prometheus Adapter to make the metrics available to the custom metrics API endpoint, and with multiple data sources feeding the Prometheus TSDB.

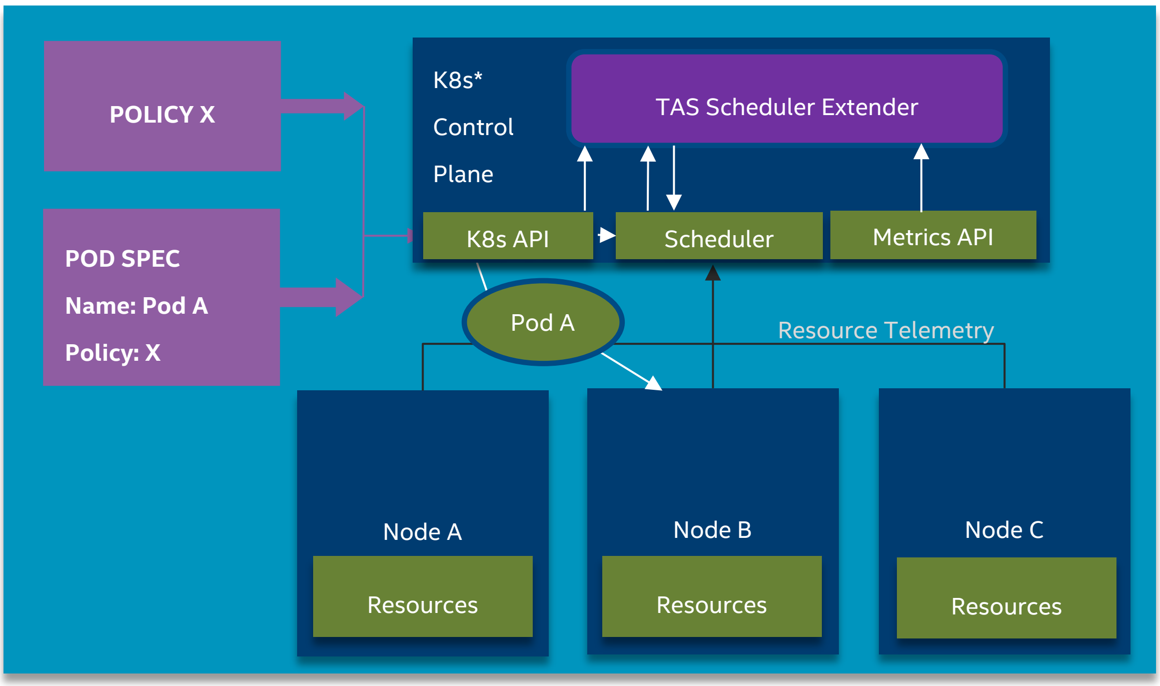


Figure 7. Telemetry Aware Scheduling for Kubernetes

At the data source layer of the solution there are multiple options such as the Prometheus Node Exporter, which has an extensive set of default and optional metrics it can collect and make available to Prometheus [Ref 37]. In addition to the Kubernetes native data, for more infrastructure-specific and in-depth data collection, tools such as `collectd` [Ref 38] and `telegraf` [Ref 39] are available. Intel has invested in plugins to these frameworks with a focus on providing access to a rich set of telemetry data in the Intel Xeon Scalable processor across its many subsystems inclusive of power, Reliability/Availability/Serviceability (RAS), Intel RDT, and accelerators.

TAS supports four policies that control the scheduling action it takes based on the telemetry data assessed:

- `deschedule`: Initiates the Pod scheduling process for all Pods on the node identified.
- `scheduleonmetric`: This policy aims to prioritize a node to be selected for scheduling based on the comparison of a metric input to a specified value.
- `dontschedule`: This policy restricts a node from being selected for scheduling.
- `label`: This policy applies labels to nodes based on the rules.

7.2 GPU Aware Scheduling

GPU Aware Scheduling (GAS) is architecturally similar to TAS. GAS is a GPU-focused scheduling extender that focuses on GPU device-level resource management by adding scheduler-level capability to perform the node-level accounting for the GPU resources. The enhancements to the scheduler support advanced use cases such as ensuring appropriate node selection when sharing a GPU with multiple containers in a multi-GPU enabled node. GAS can also use GPU-specific telemetry data to inform scheduling decisions.

Unlike TAS, GAS has an additional function related to the annotation of Pods post node selection. Annotations include information such as timestamping and noting the GPU cards that were selected for containers in Pods. At the node resource assignment phase, these annotations are used by the Intel Kubernetes GPU device plugins to support resource assignment.

The same Prometheus TSDB, Prometheus Adapter, and Kubernetes Custom Metrics API components are used as with TAS, to get information to the scheduler extender. GPU-specific data that is collected with Intel® oneAPI Level Zero (Level Zero) Sysman GPU plugin for `collectd` or Intel® XPU Manager (Intel® XPUM) Prometheus exporter. These exporters can collect metrics such as available GPU memory or GPU engine utilization and relay up the observability stack.

GAS has several user-definable policy options. It can trigger actions based on collected GPU data such as removing a node from a scheduling decision due to insufficient availability of resources on a per-GPU implementation basis, pack or distribute Pods across node GPUs, and assign a group of containers to the same GPU.

At time of writing, the enablement is supported on the Intel® Data Center GPU Flex Series and the Intel® Data Center GPU Max Series.

8 Intent Driven Orchestration

The Kubernetes community promotes the declarative nature of the API, where in the resource orchestration case, users declare their desired resources and Kubernetes manages the deployment to get those resources allocated. The different QoS models (section 3.1) are an example of the flexibility in the declared request. Class-based API advancements such as for Intel RDT enablement (section 5.5.2) show how capabilities can be further grouped and simplified for users.

All this wonderful work supports a vast variety of deployment use cases. However, all of it has a significant drawback in that the user must make a resource orchestration choice. That choice could be aligned with the `BestEffort` QoS model where the choice is not to specify resources. In this case, the first order resource consideration for a cluster, i.e., the type of infrastructure underpinning the cluster, has significant impact on the behavior of workloads. An application behaves differently if it is running on the latest Intel Xeon Scalable processor versus a different class of processor or indeed on a processor that is many generations older. A user using a 4th Gen Intel Xeon Scalable processor and specifying a request and limit of two CPUs sees different performance than if they are using a 3rd Gen Intel® Xeon® Scalable processor with the same request and limits.

The other Kubernetes QoS models offer the ability to declare more fine-grained requirements of resources from the cluster, and this approach can demonstrably show improvements in workload characteristics.

Some of the challenges with these declarative approaches are:

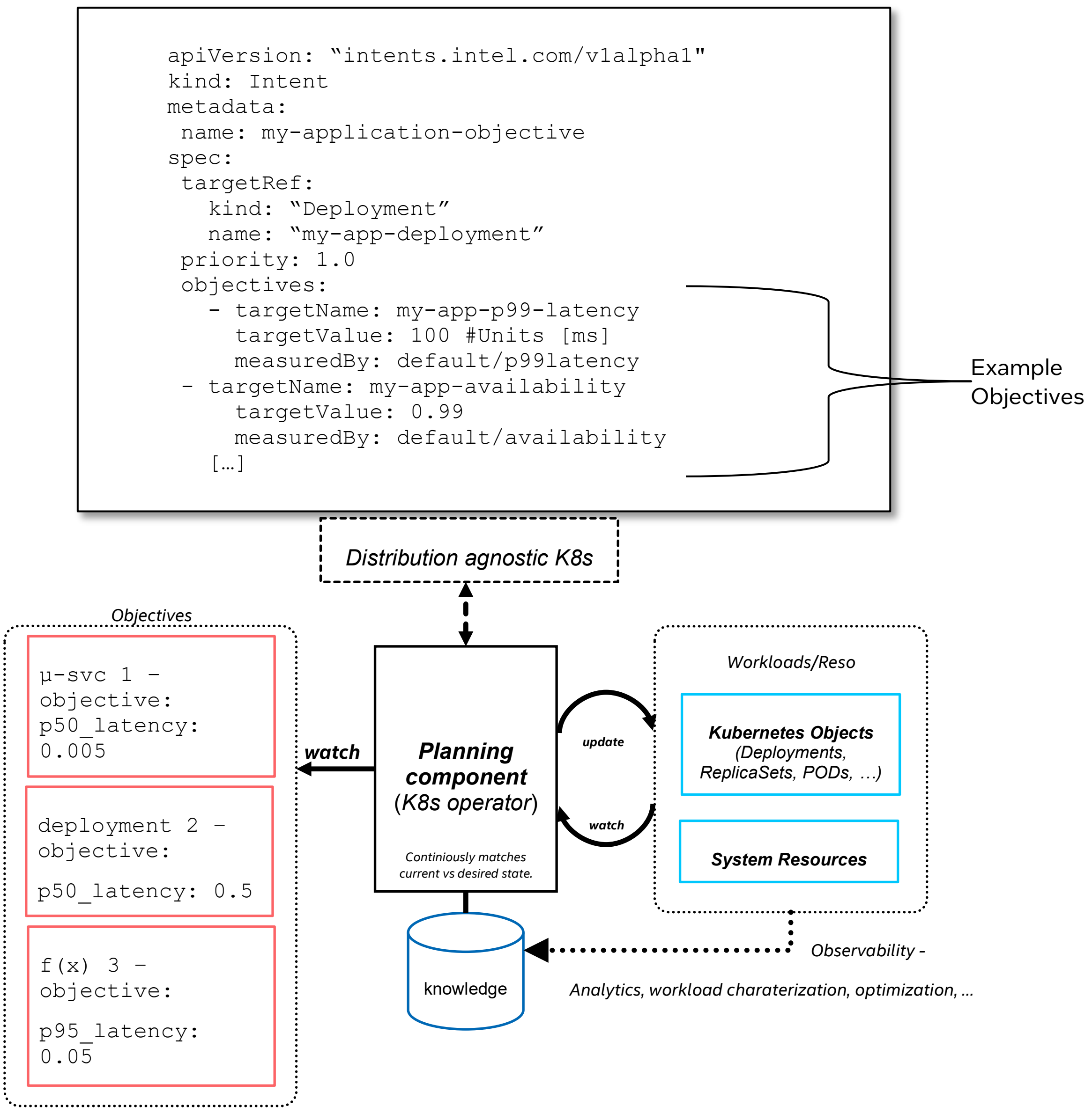
1) They do not capture the actual workload objective, i.e., these are not requests for one or more workload KPIs.
2) They only represent a syntactically portable approach between Kubernetes clusters. I.e., the same Pod spec can run on two clusters, but there are no assurances that the workload will perform the same. Aspects of semantic portability are not catered for.
3) The burden of rightsizing the resource requests is placed on the application developer.
4) The resource request cannot adapt to changing conditions within a cluster.

Intent-Driven Orchestration (IDO) is an alternative approach that addresses these challenges. At time of writing, this is a new proposal recently shared with the Kubernetes community. The initial goals for the project's first release are to solicit feedback on the API and approach and to establish community participant interests in collaborating. For the most up-to-date information on IDO, refer to the GitHub site [Ref 40].

The IDO v0.1.0 API provides options for application KPI-based objectives to be specified inclusive of performance, latency, availability, and power objectives. Objectives are all defined with a target value and a mechanism by which the IDO planner can ingest observability data that is monitoring that KPI. When this interface is used, there is no requirement on the Pod spec to specify resources.

The IDO Planner parses the objectives for the Pod and through multiple mechanisms determines the initial resource requests. The Pod spec is modified via a set of actuators accordingly after which the Kubernetes scheduling and resource allocation routines (including those mentioned in this paper) are activated to reposition the workload. This process is iterated to determine the best resource fit for the Pod given its intent and the current state of the cluster.

The initial reference IDO implementation depicted in Figure 8 is an example of the closed-loop automation system that can benefit from the goodness of the Kubernetes state machine. With the Kubernetes ability with desired state management, IDO can actively adjust to a minimal viable resource footprint to support an application.



```
apiVersion: “intents.intel.com/v1alpha1"
kind: Intent
metadata:
 name: my-application-objective
spec:
 targetRef:
   kind: “Deployment”
   name: “my-app-deployment”
 priority: 1.0
 objectives:
   - targetName: my-app-p99-latency
     targetValue: 100 #Units [ms]
     measuredBy: default/p99latency
 - targetName: my-app-availability
     targetValue: 0.99
     measuredBy: default/availability
   […]
```

Example Objectives
Distribution agnostic K8s
Objectives
µ-svc 1 - objective: p50_latency: 0.005
deployment 2 - objective: p50_latency: 0.5
f(x) 3 - objective: p95_latency: 0.05
watch
Planning component (K8s operator)
Continiously matches current vs desired state.
update
watch
Workloads/Reso
Kubernetes Objects (Deployments, ReplicaSets, PODs, …)
System Resources
knowledge
Observability -
Analytics, workload charaterization, optimization, ...

Figure 8. Intent Driven Orchestration for Kubernetes

9 Benefits of Advanced Resource Orchestration

The tests noted in this section were carried out to demonstrate the potential value of some of the capabilities mentioned in this document.

9.1 Test Environment

A 4th Gen Intel Xeon Scalable processor, the Intel® Xeon® Platinum 8480+ processor, was used for the Kubernetes control plane node and for each of the three worker nodes. Figure 9 depicts the cluster topology where the single Kubernetes control plane node was executed from as well as the traffic generator. The three worker nodes had identical configurations.

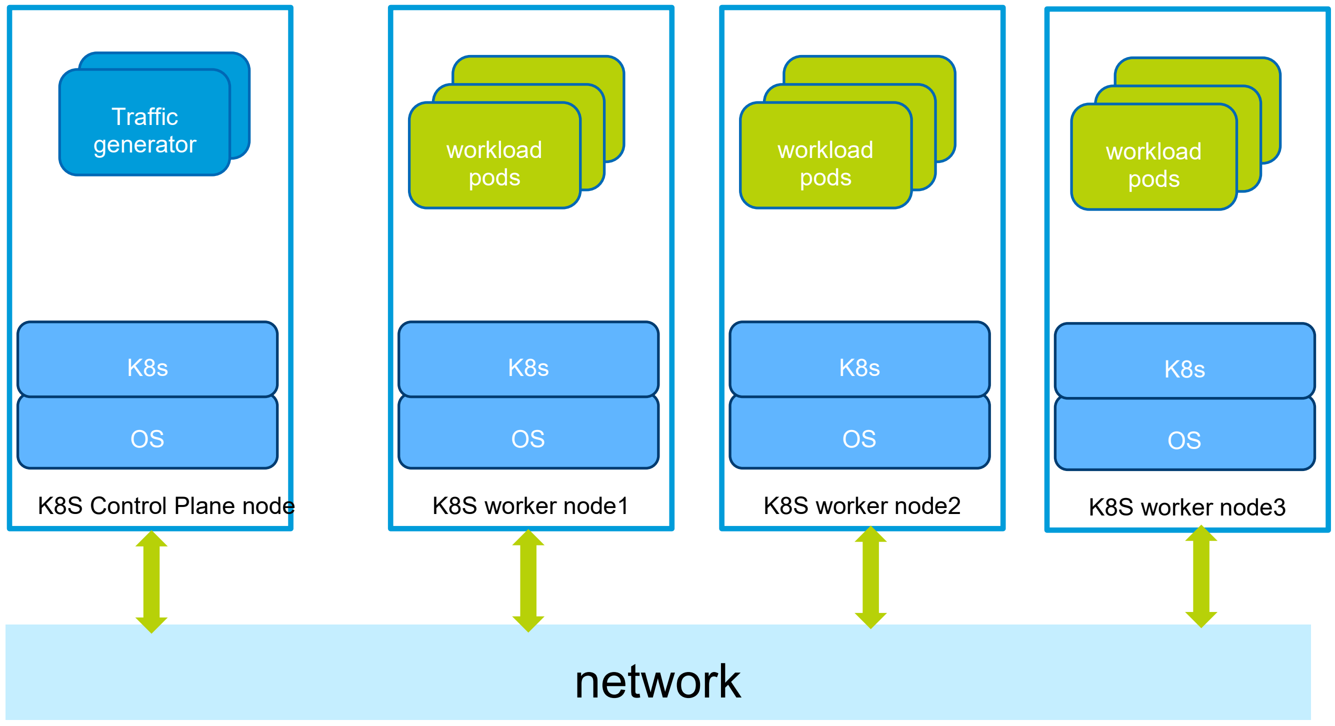


Figure 9. Kubernetes Cluster Topology for Benchmarking

A detailed description of the system configuration is provided in Table 2.

Table 2. Benchmarking System Configuration

Name	Description
Name	r028s007
Time	Mon Dec 19 10:46:54 AM UTC 2022
Manufacturer	Intel Corporation
Product Name	Archer City
BIOS Version	EGSDCRB1.SYS.8901.P01.2209200243
OS	Ubuntu 22.04.1 LTS
Kernel	5.15.0-56-generic
Microcode	0x2b0000a1
IRQ Balance	Enabled
CPU Model	Intel(R) Xeon(R) Platinum 8480+
Base Frequency	2.0GHz
Maximum Frequency	3.8GHz
All-core Maximum Frequency	3.0GHz
CPU(s)	224
Thread(s) per Core	2

Name	Description
Core(s) per Socket	56
Socket(s)	2
NUMA Node(s)	2
Prefetchers	L2 HW, L2 Adj., DCU HW, DCU IP
Turbo	Enabled
PPIN(s)	28bff24bb26f835f,28bff34b4b3fa975
Power & Perf Policy	Performance
TDP	350 watts
Frequency Driver	intel_pstate
Frequency Governor	performance
Measured Frequency (MHz)	2.75
Max C-state	9
Installed Memory	1024GB (32x32GB DDR5 4800 MT/s [4400 MT/s])
Huge Pages Size	2048 KB
Transparent Huge Pages	madvise
Automatic NUMA Balancing	Enabled
Ethernet Network Adapter Summary	2x Intel® Ethernet Network Adapter E810-CQDA2
Drive Summary	1x 54.9G INTEL SSDPEK1A058GA

9.2 Google Microservices Benchmark

The Google Microservices (GMS) benchmark [Ref 41] is a demo application of a microservices-based online boutique. It is an e-commerce application where users can browse items, add them to the cart, and purchase them. GMS is composed of 12 microservices implemented in Go, C#, Node.js, Python, and Java. The microservices are interconnected with gRPC, and a single instance of a GMS deployment consists of approximately 300 containers.

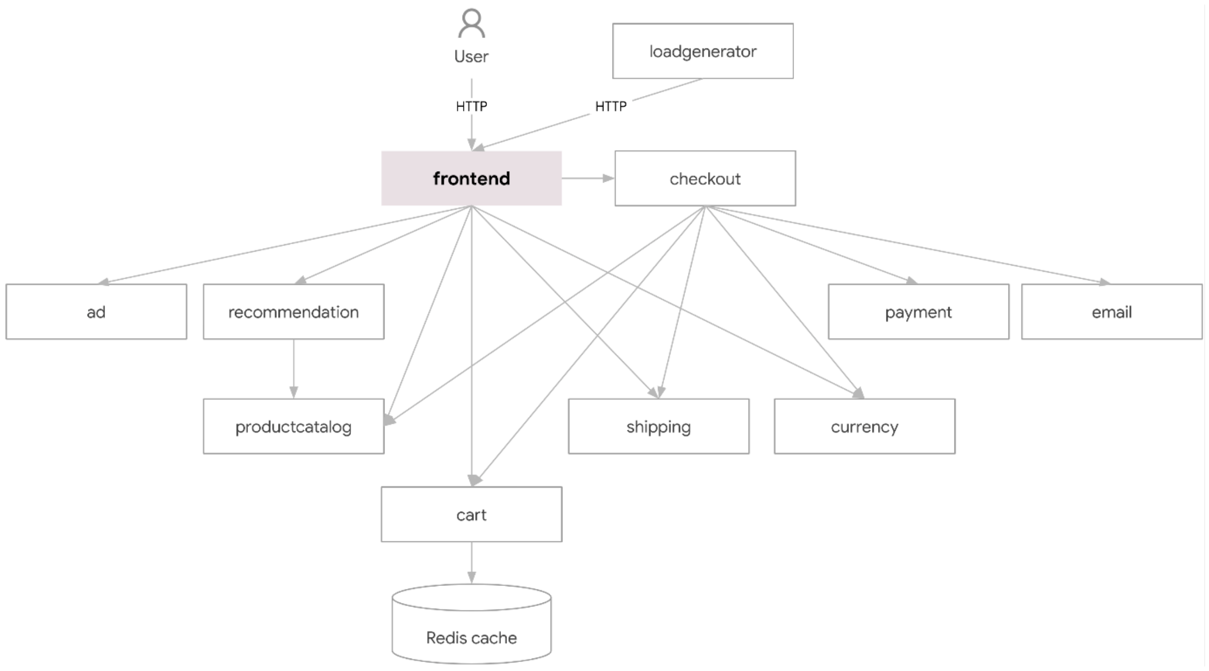


Figure 10. Google Microservices Benchmark architecture diagram[2]

[2] Source: https://github.com/GoogleCloudPlatform/microservices-demo/raw/main/docs/img/architecture-diagram.png

GMS was tested in four different configurations captured in Table 3. Configuration 1 is the default configuration and represents the out-of-the-box performance. The changes with config 2 focus on increasing the number of instances of GMS from one to two instances. This config 2 is used as the baseline for relative performance improvements demonstrated. Config 3 and config 4 focus exclusively on deployment reconfigurations, i.e., Pod spec changes only, combined with the addition of the CPU Control Plane Manager (CtlPlane) to the Kubernetes installation.

Table 4 indicates how the different GMS microservices were grouped to deploy along namespace boundaries (GMS1, GMS2, and GMS3). Table 5 captures how the default Kubernetes `best-effort` QoS model deployment for GMS was updated to have a combination of `best-effort` and `guaranteed` QoS deployment allocations for the different microservices. This new configuration was facilitated by the addition of the CPU Control Plane Manager to the Kubernetes installation.

Table 3. Google Microservices Workload Configurations for Benchmarking

Host Component	Config 1 (Default)	Config 2 (Baseline)	Config 3 (Baseline + CtlPlane)	Config 4 (Baseline + Ctlplane + 4 namespaces)
OS Distribution	Ubuntu 22.04.1 LTS			
Kernel	5.15.0-56-generic			
Ethernet Network Adapter Driver	ice 5.15.0-56-generic			
Compiler	Golang 1.18.6			
GLIBC	Ubuntu GLIBC 2.35-0ubuntu3.1			
Kubernetes (K8s)	1.25.6			
K8s CRI	Containerd 1.6.8 Control Plane 0.1.1			
K8s CNI	Calico v3.23.3			
Workload Component	GMS 1.0			
Traffic generator	open loop wrk2 included in GMS: mixed-workload.lua			
workload instance #	1	2	2	4
namespaces	1	1	2	4
replica # (each instance)	60	30	See Table 4	Half the values indicated in Table 4
QoS Setting	`BestEffort`	`BestEffort`	See Table 5	See Table 5
wrk2 instance #	1	2	2	4
wrk2 thread # (each instance)	20			
wrk2 connection # (each instance)	1,600			
wrk2 input rate # (each instance)	60,000			

Table 4. Google Microservices Replica Count for Two Namespace Deployment Configurations

GMS1								
	adserv	currency	email	payment	product	shipping	redis	total
ratio	0.05	0.06	0.04	0.04	0.09	0.04	0.00	0.31
replicas	15	18	12	12	27	12	1	97

GMS2								
	cart	checkout	recomm					total
ratio	0.10	0.11	0.12					0.32
replicas	30	35	36					101

GMS3								
	frontend							total
ratio	0.32							0.32
replicas	100							100

Table 5. Google Microservices Mixed Deployment Model QoS Configuration (best-effort with guaranteed QoS)

µService	`BestEffort`	`guaranteed` QoS (1 core; 512 Mi Memory)
frontend	no	yes
adservice	yes	no
currency	no	yes
emailservice	yes	no
payment	yes	no
productcatalogue	yes	no
redis	yes	no
shipping	yes	no
cartservice	yes	no
checkoutservice	no	yes
recommendation	no	yes

The results of the testing are captured in Figure 11. The Config2 performance data is for two instances of GMS deployed on three worker nodes, with the other data points normalized against this base point. After deploying GMS with the aid of the Intel CPU Control Plane Manager with the Pod spec tuned to request a mix of `guaranteed` QoS for some Pods and `best-effort` NUMA-isolated deployment for other Pods, and four instances deployed on the same number of worker nodes combined with a reduction in the number of replicas (to keep the number of containers approximately constant between test configurations), the throughput increased ~x2.06 over baseline. The average CPU utilization also jumped from the baseline reading of ~30% to ~75%, and average use of local DRAM increased from ~46% to ~81%, demonstrating more efficient use of the processor cores and memory.

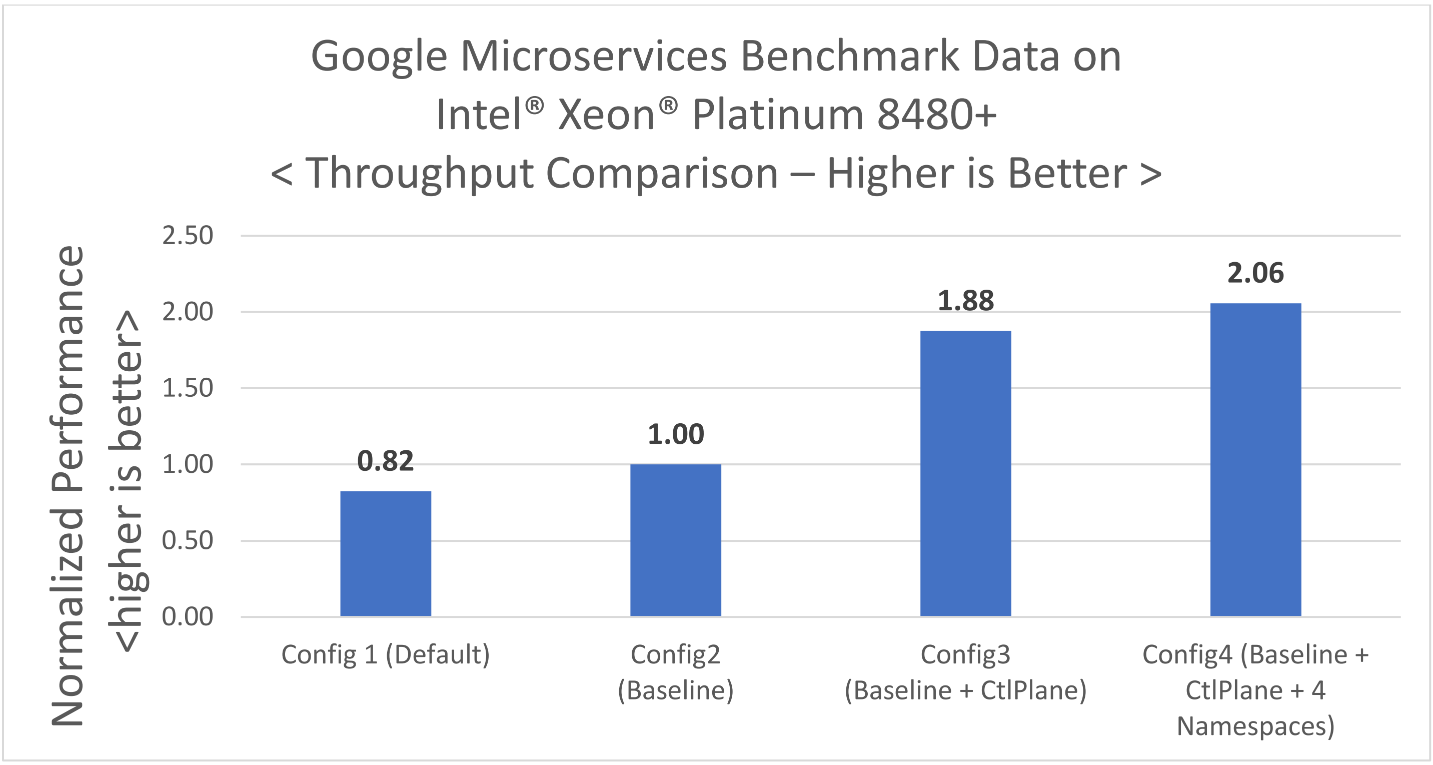


Figure 11. Google Microservices Benchmark Performance with Intel CPU Control Plane Manager on 4th Gen Intel Xeon Scalable Processor

9.3 Death Star Bench

Death Star Bench (DSB) [Ref 42] is an open source, microservices benchmark suite with several applications written in modern, cloud native architecture, originally developed by Cornell University. The hotelReservation application in Death Star Bench mimics a typical microservice workload of a hotel booking system. It is written in golang and uses gRPC-go for inter-microservice communication. It is used to show various characteristics of microservice workloads in cloud-native environment.

In this test environment, the Hotel Reservation benchmark was deployed in four different configurations. The default out-of-the box Config 1 testing indicated a bottleneck in database accesses. Config 2 was chosen as the baseline configuration and deployed with four database instances. Config 3 added the CPU Control Plane Manger and Config 4 extended Config 3 with the addition of Multus to the Kubernetes installation to enable multiple network connections. Table 6 captures the different configuration settings that were tested.

Table 6. Death Star Bench Testing Configurations

Host Component	Config 1 (Default)	Config 2 (Baseline)	Config 3 (Baseline + Ctlplane)	Config 4 (Baseline + Ctlplane + Multus)
OS Distribution	Ubuntu 22.04.1 LTS			
Kernel	5.15.0-56-generic			
Ethernet Network Adapter Driver	ice 5.15.0-56-generic			
Compiler	Golang 1.18.6			
GLIBC	Ubuntu GLIBC 2.35-0ubuntu3.1			
Kubernetes(K8s)	1.25.6			
K8s CRI	Containerd 1.6.8 Control Plane 0.1.1			
K8s CNI	Cilium: 1.11.5 & Multus v3.8			
Workload Component	DeathStarBench hotelReservation 1.0			
Image	lianhao/dsbpp_hotel_reserve:1.0			
golang	1.17.3			
gRPC-go	1.10			
Consul	1.9.2			
Memcached	1.6.8			
MongoDB	4.4.3			
Traffic generator	open loop wrk2 included in DSB: mixed-workload_type_1.lua			
workload instance #	1	4	4	4
Database Instances	1	4	4	4
replica # (each instance)	24	6	6	6
wrk2 instance #	1	4	4	4
wrk2 thread # (each instance)	48			
wrk2 connection # (each instance)	1920			
wrk2 input rate # (each instance)	120000			

The out-of-the-box default performance data for one instance of DSB deployed on three worker nodes was shown to be significantly slower than a deployment with four DSB instances and four databases. The Config 2 baseline captured an average CPU load of ~70% and an average use of local DRAM at ~48%. With Config 4, deploying DSB with the aid of the Intel CPU Control Plane Manager with the Pod spec tuned to request a mix of `guaranteed` QoS for some Pods and `best-effort` NUMA-isolated deployment for other Pods combined with the use of multiple network interfaces, the throughput increased ~1.8x over the baseline. The average CPU utilization also increased to ~80% and the average use of local DRAM increased to ~88%, demonstrating more efficient use of the processor cores and memory.

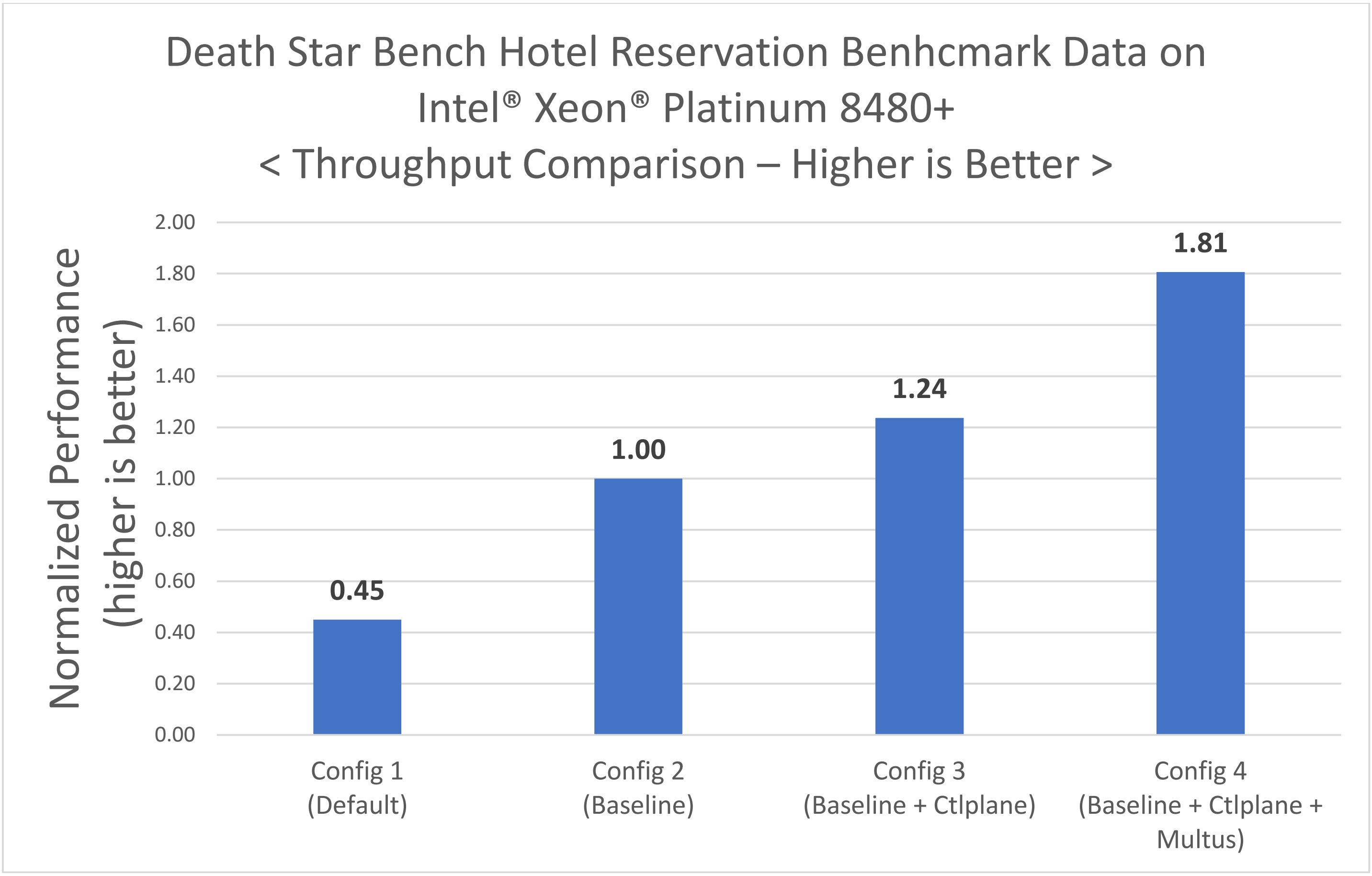


Figure 12. Death Star Bench Hotel Reservation Performance with Intel CPU Control Plane Manager on 4th Gen Intel Xeon Scalable Processor

At the time of writing, the data was not available for CRI-RM. From a performance perspective, it is anticipated to deliver comparable results with these workloads. Users looking to choose between these options are advised to focus their attention on the differences in user and administrator controls as well as their requirements on the dynamic configurations needed to support a selection decision.

10 References

1. https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/
2. https://kubernetes-sigs.github.io/node-feature-discovery/stable/get-started/index.html
3. https://builders.intel.com/docs/networkbuilders/node-feature-discovery-application-note.pdf
4. https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/
5. https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/
6. https://kubernetes.io/docs/tasks/administer-cluster/memory-manager/
7. https://github.com/intel/intel-device-plugins-for-kubernetes
8. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html
9. https://networkbuilders.intel.com/solutionslibrary/queue-management-and-load-balancing-on-intel-architecture
10. https://www.intel.com/content/www/us/en/developer/articles/technical/scalable-io-between-accelerators-host-processors.html
11. https://www.computeexpresslink.org/
12. https://github.com/intel/CPU-Manager-for-Kubernetes
13. https://networkbuilders.intel.com/solutionslibrary/cri-resource-manager-topology-aware-resource-assignment-in-kubernetes-technology-guide
14. https://github.com/containerd/nri/
15. https://github.com/containerd/containerd/releases/tag/v1.7.0-beta.1
16. https://github.com/cri-o/cri-o/releases/tag/v1.26.0

17. https://intel.github.io/cri-resource-manager/stable/docs/policy/topology-aware.html
18. https://intel.github.io/cri-resource-manager/stable/docs/policy/static-pools.html
19. https://intel.github.io/cri-resource-manager/stable/docs/policy/container-affinity.htm
20. https://github.com/intel/CPU-Manager-for-Kubernetes
21. https://github.com/intel/cri-resource-manager/blob/master/sample-configs/balloons-policy.cfg
22. https://intel.github.io/cri-resource-manager/stable/docs/policy/podpools.html
23. https://intel.github.io/cri-resource-manager/stable/docs/policy/container-affinity.html
24. https://docs.kernel.org/x86/resctrl.html
25. https://www.kernel.org/doc/html/latest/x86/resctrl.html
26. https://github.com/intel/goresctrl/blob/v0.2.0/doc/rdt.md#configuration
27. https://github.com/containerd/containerd/blob/main/docs/man/containerd-config.toml.5.md
28. https://github.com/intel/goresctrl/blob/main/doc/rdt.md#examples
29. https://intel.github.io/cri-resource-manager/stable/docs/policy/rdt.html
30. https://builders.intel.com/docs/networkbuilders/power-management-technology-overview-technology-guide.pdf
31. Advanced Configuration and Power Interface (ACPI) specification: https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/
32. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
33. https://www.kernel.org/doc/html/latest/admin-guide/pm/intel_uncore_frequency_scaling.html
34. https://github.com/intel/kubernetes-power-manager
35. https://github.com/intel/power-optimization-library
36. https://github.com/intel/platform-aware-scheduling
37. https://github.com/prometheus/node_exporter#collectors
38. https://collectd.org/
39. https://github.com/influxdata/telegraf
40. https://github.com/intel/intent-driven-orchestration
41. https://github.com/GoogleCloudPlatform/microservices-demo
42. Death Star Bench: https://www.csl.cornell.edu/~delimitrou/papers/2019.asplos.microservices.pdf

0223/DN/WIT/PDF 764937-002US