TECHNOLOGY GUIDE

Intel Corporation

CRI Resource Manager – Topology-Aware Resource Assignment in Kubernetes

Authors

Alexander Kanevskiy

Krisztian Litkey

Markus Lehtonen

Antti Kervinen

1 Introduction

Kubernetes has a limited view of the hardware topology details of the worker nodes in a cluster, which may cause unexpected performance degradation or loss of potential performance benefits. For example, Kubernetes would not be aware of the hierarchy of memory controllers, high/low clock frequency CPU cores, and some types of memory.

The CRI Resource Manager, an add-on plugging in between Kubernetes and the container runtime, provides a method of improving the performance of containerized workloads on bare metal Kubernetes cluster nodes. It provides optimized workload placement for worker nodes with multi-tiered memory or Intel® sub-NUMA clustering (SNC) using technologies such as Intel® Resource Director Technology (Intel® RDT), Intel® Speed Select Technology (Intel® SST), Linux Block IO controller, and more.

This document provides instructions for the installation and configuration of CRI Resource Manager, describes its functionality, and discusses possible usage scenarios. It is intended for bare-metal cluster operators wanting to capitalize on the full potential of the latest Intel® Xeon® Scalable processors.

This document is part of the Network Transformation Experience Kit, which is available at https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits.

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	April 2021	Initial release.

1.1 Terminology

Table 1. Terminology

ABBREVIATION	DESCRIPTION
CMK	CPU Manager for Kubernetes
CRI	Container Runtime Interface
CRI-RM	CRI Resource Manager
K8s	Kubernetes
NFV	Network Function Virtualization
PMEM	Persistent Memory
QoS	Quality of Service
RDT	Intel® Resource Director Technology (Intel® RDT)
SIMD	Single Instruction Multiple Data
SNC	Intel® Sub-NUMA Clustering
TDP	Thermal Design Power
VNF	Virtual Network Function

1.2 Reference Documentation

Table 2. Reference Documents

REFERENCE	SOURCE
CRI Resource Manager User Documentation	https://intel.github.io/cri-resource-manager
CPU Management - CPU Pinning and Isolation in Kubernetes* Technology Guide	https://builders.intel.com/docs/networkbuilders/cpu-pin-and-isolation-in-kubernetes-app-note.pdf
Kubernetes Dynamic Admission Control	https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

2 Overview

2.1 Challenges Addressed

Kubernetes currently treats CPU, memory, and hugepages as first-class resources, or compute resources in the Kubernetes terminology. The Kubernetes control plane implements fairly sophisticated accounting and arbitration algorithms for assigning these resources to workloads. GPUs, FPGAs, various computation accelerators, and other peripherals are treated as opaque non-sharable resources, known as extended resources in the Kubernetes terminology. Accounting and arbitration of these is fairly simple, with every instance of an extended resource always allocated exclusively, i.e., to a single container at a time.

Kubernetes implements a number of optimizations to improve the performance of high priority workloads. It supports CPU pinning and isolation, the exclusive allocation of a dedicated subset of CPUs to a single workload. It also implements best-effort alignment of CPUs, GPUs, and other peripheral devices, attempting to minimize the I/O latency when multiples of these are allocated to a single workload. It can also pin critical workloads to a single NUMA node to reduce the latency of memory access.

The optimizations present in Kubernetes can deliver big improvements for a number of workloads. However, Kubernetes' naively simplified view of the hardware both prevents these from reaching their full potential and leaves room for further optimizations, especially for special-purpose performance-critical workloads running on dedicated bare metal clusters.

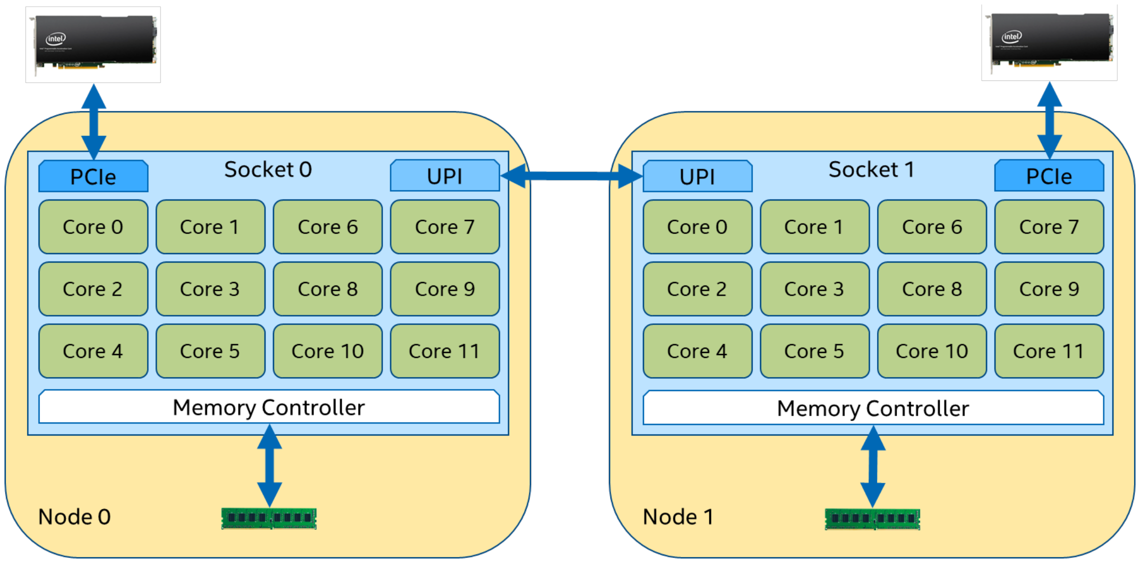

Figure 1. Kubernetes' View of System Resources

Modern server-grade hardware is complex. To paraphrase the once famous slogan of a since-deceased internet pioneer company, we've come from a world where the 'network is the computer' to one where 'the computer is a network'. And not even a homogeneous one.

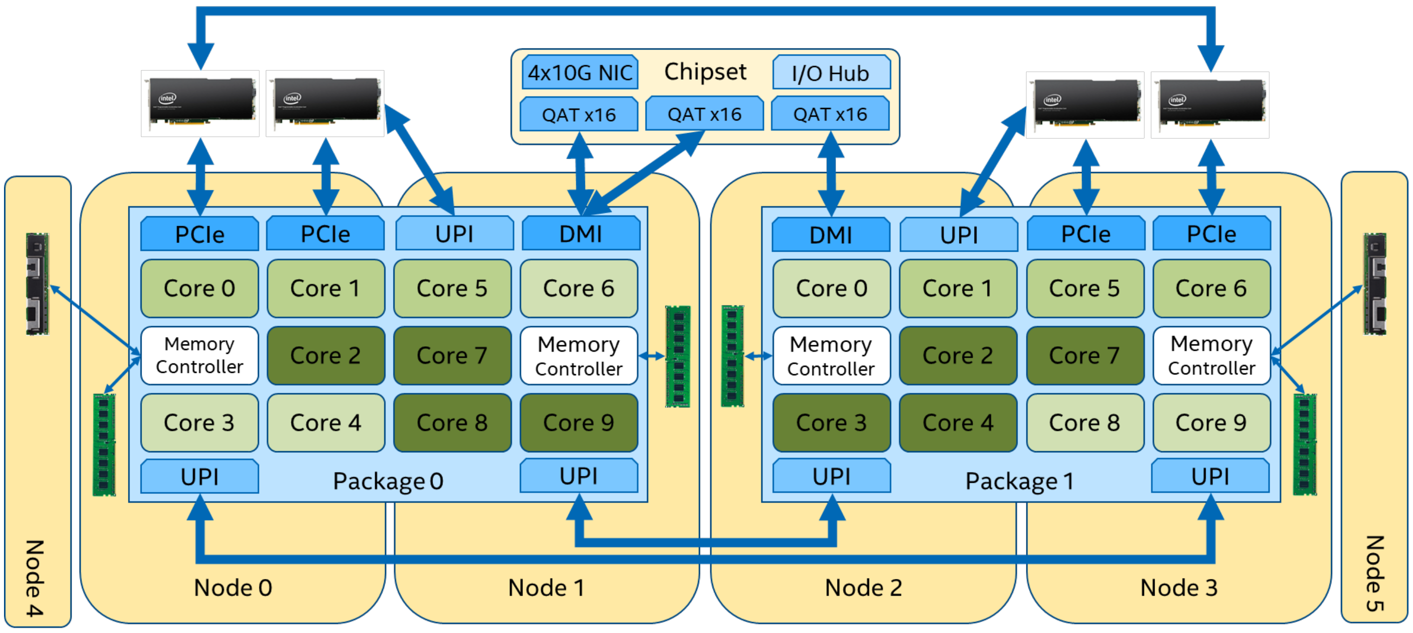

Figure 2. True Topology of System Resources

Kubernetes worker nodes typically run on machines with multiple sockets, each potentially containing multiple dies. Each die or socket has numerous CPU cores and multiple memory controllers. There is an interconnect fabric between the sockets or dies, enabling data transfer between their cores and memory controllers. There is also a hierarchy of buses present where additional peripherals, GPUs, FPGAs, and other devices are attached. All these form a complex network or mesh, with limited data transfer bandwidth and varying latency along the various paths. In that network,

- The cost of accessing any memory from any core is not uniform.
- The cost of accessing any device from any core is not uniform.

Or to put this in more computational terms,

- Latency of accessing memory or I/O devices varies.
- Bandwidth available for accessing memory or I/O devices varies.

In addition, the performance of CPU cores may be different from each other.

In such a system, the placement of a workload, in other words the choice of CPU cores to run the workload, the memory controllers used to allocate memory for the workload, the choice of I/O devices to assign to the workload, and especially the combination of these, can have a significant impact on how efficiently the workload can perform both in terms of throughput and latency. Having a more realistic understanding of the hardware to make better informed intra-node workload placement decisions can improve performance.

There are also additional sources for potential optimizations. For example, the last level cache is a shared resource within a die or a socket. Modern hardware offers a control mechanism over how much of that cache workloads can use. By limiting the size of the cache for low priority workloads and dedicating a larger portion for high priority workloads, often better performance guarantees can be provided for the latter. Analogous control mechanisms exist and can be applied the same way to guaranteeing or prioritizing available memory bandwidth. Likewise, by using the mechanism offered by the Linux kernel to throttle disk I/O, low priority workloads can be deprioritized and more disk I/O, or other bus bandwidth, can be set aside for high priority ones.

There are CPUs with special instructions that might influence the TDP budget and the maximum concurrent clock frequency budget available for the rest of the core within the same die or socket. This introduces another dimension and potential source for optimizations for workload placement: workload cross-effects and dynamic behavior.

Collecting metrics for monitoring and identifying trending shifts in the dynamic runtime behavior of workloads, then reshuffling their placement in a more optimal way for the current behavior after clear changes are detected, can sometimes result in significant performance improvements.

CRI Resource Manager provides a multitude of mechanisms for improving the intra-node resource allocation and thereby addressing the potential challenges described above. In addition to general optimizations, it also uses many technologies found in the latest Intel® Xeon Scalable processors. Table 3 below shows an overview of the available features.

Table 3. Feature Comparison

FUNCTIONALITY	VANILLA KUBERNETES	CRI RESOURCE MANAGER
CPU Pinning	Limited (with CPU Manager)	Yes
Topology-Aware Scheduling		
Memory Tiering	No	Yes
Intel® sub-NUMA Clustering (SNC)	No	Yes
Intra-Node Container Affinity	No	Yes
Intel® Resource Director Technology (Intel® RDT)	No	Yes
CPU Core Priority Detection		
Isolated CPUs (isolcpus)	No	Yes
Intel@ Speed Select Technology Base Frequency	No	Yes
Intel@ Speed Select Technology Core Power	No	Yes
Block IO cgroup control	No	Yes
Dynamic, centralized node/group configuration	No	Yes

2.2 Use Cases

2.2.1 Topology-Aware Resource Alignment

On server-grade hardware, the CPU cores, memory, I/O devices, and other peripherals form a rather complex network together with the memory controllers, the I/O bus hierarchy, and the CPU interconnect. When a combination of these resources is allocated to a single workload, the performance of that workload can vary greatly depending on how efficiently data is transferred between them, or in other words on how well the resources are aligned.

There are a number of inherent architectural properties of server-grade hardware that, unless properly taken into account, can cause resource misalignment and workload performance degradation. There are a multitude of CPU cores available to run workloads. There are a multitude of memory controllers that these workloads can use to store and retrieve data from main memory. There are a multitude of I/O devices attached to a number of I/O buses that the same workloads can access. The CPU cores can be divided into a number of groups, with each group having different access latency and bandwidth to each memory controller and I/O device.

If a workload is not assigned to run with a properly aligned set of CPU, memory, and devices, it will not be able to achieve optimal performance. Given the previously mentioned idiosyncrasies of hardware, allocating a properly aligned set of resources for optimal workload performance requires identifying and understanding the multiple dimensions of access latency locality present in the hardware, or in other words hardware topology awareness.

2.2.2 Noisy Neighbors

The cloud is a multi-tenant environment, which means that a single architecture hosts multiple customers' applications and data. In practice, however, workloads running on the same server can and usually do interfere with each other. If they run on sibling hyper-threads, they use the same physical CPU core. If they run on different physical CPU cores, they can share the same last level cache, memory controller, and I/O bus bandwidth. Even if workloads run on different physical CPU sockets and use different cache and memory channels, they can share CPU interconnect bandwidth, the same storage devices, or I/O bus.

Some workloads try to use a disproportionately large amount of the shared hardware resources with respect to both the number of total workloads and their own relative importance. These workloads are collectively referred to as 'noisy neighbors.' Some workloads in a cluster are of critical importance, for example a database used to store and serve data for tens or hundreds of other workloads in the cluster, or a virtualized network service responsible to deliver data to thousands or tens of thousands of customers simultaneously. When such critical workloads are co-located with one or more noisy neighbors, it is important to ensure that the critical workloads get their share of resources, CPU, cache, memory, I/O bandwidth, and the like for smooth operation at any point in time.

2.2.3 Latency-Critical Workloads

An increasing number of network-intensive latency-sensitive workloads have moved to run on cloud infrastructure. Some of these are regular applications, such as content delivery network servers or ultra-high-performance web servers. Advances in hardware have made it feasible to implement network functions on platforms based on general purpose processors. As a result, an increasing number of these workloads are virtual network functions (VNFs), which are used to virtualize network services, routers, firewalls, and load balancers, etc., and which traditionally have required running on proprietary hardware.

The ongoing commercial rollout of large-scale publicly available 5G networks has only accelerated the trend of adopting cloud infrastructure as the underlying platform for hosting network function virtualization (NFV). While in some classes of workloads abnormal latency variation and throughput instability is merely a nuisance, in workloads that provide critical network infrastructure that nuisance quickly becomes something that can even prevent regulatory approval.

Since the cloud is by nature a multi-tenant environment with several applications being co-hosted on the same hardware, it is inherently prone to applications interfering with each other. Reliable and predictable performance is a critical requirement for NFV. Therefore, for NFV special care and consideration is necessary when running on a multi-tenant cloud, to protect the latency and other performance critical workloads from interference by other co-hosted applications.

2.2.4 CPU Clock Speed Throttling

Somewhat similar to the 'noisy neighbors,' some workloads may inadvertently degrade the performance of neighboring or co-located workloads by inflicting throttling of CPU clock frequency. This may be, for example, a side effect of using complex SIMD instructions that cause the maximum clock speed of the executing CPU core and its neighboring CPU cores to be throttled. This, in turn, may result in unwanted performance degradation of neighboring CPU-intensive workloads.

There are a few strategies that can work to mitigate these negative side effects. Spreading the disruptive workloads evenly over multiple clock- and TDP-domains, or in other words CPU cores and sockets, to minimize their effect can often be a good mitigating strategy. This usually works well as long as the number of offending workloads stays below a critical threshold. After that threshold is reached, if, despite the spread out, enough of the offending workloads are co-located to cause significant clock throttling, the alternative strategy of squeezing these workloads to as few clock- and TDP-domains as possible can prove to be a better strategy and bring observable benefits to the other workloads, both in terms of absolute performance and consistency.

2.3 Technology Description

CRI Resource Manager has three separate software components. The resource manager daemon is the main component responsible for workload management and communications with kubelet and the container runtime. It is supplemented by an optional node agent handling communication with the Kubernetes control plane, and an optional webhook responsible for annotating pods with side-channel information about their resource requests. Figure 3 shows the different components and their interaction with the Kubernetes cluster.

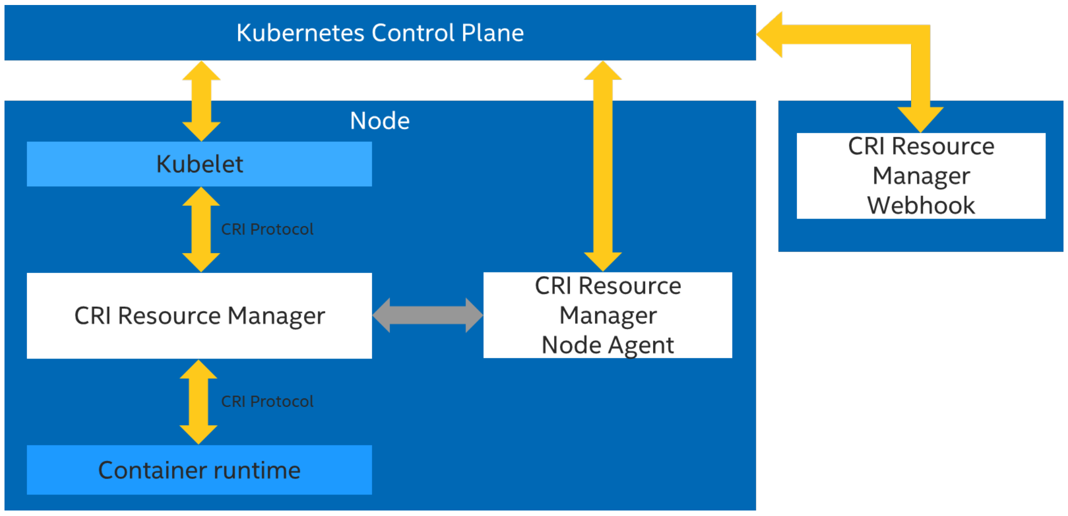

Figure 3. Components of CRI Resource Manager

2.3.1 CRI Resource Manager

CRI Resource Manager (CRI-RM) is a pluggable add-on for controlling how much and which resources are assigned to containers in a Kubernetes cluster. It is an add-on because you install it in addition to the normal selection of your components. It is pluggable since you inject it on the signaling path between two existing components with the rest of the cluster unaware of its presence. CRI-RM plugs in between the Kubernetes node agent (kubelet) and the container runtime running on the node (for example, containerd or cri-o). It intercepts CRI protocol requests from the kubelet by acting as a non-transparent proxy towards the runtime. Proxying by CRI-RM is non-transparent in nature because it usually alters intercepted protocol messages before forwarding them.

CRI-RM keeps track of the states of all containers running on a Kubernetes node. Whenever it intercepts a CRI request that results in changes to the resource allocation of any container (container creation, deletion, or resource assignment update request), CRI-RM runs one of its built-in policy algorithms. This policy makes a decision about how the assignment of resources to containers should be updated. The policy can make changes to any container in the system, not just the one associated with the intercepted CRI request. After the policy has made its decision, any messages necessary to adjust affected containers in the system are sent, and the intercepted request is modified according to the decisions and is relayed.

There are several policies available within CRI-RM, each with a different set of goals and implementing different hardware allocation strategies. Only a single policy can make decisions at any given time, the one that has been marked active by the administrator in the CRM-RM configuration.

Although most of the policy decisions are enforced in CRI-RM by simply altering or generating CRI protocol messages, CRI-RM also contains support for extra policing beyond what is possible to control using the CRI runtime alone. For such extra functionality, CRI-RM contains a number of dedicated components, generally referred to as controllers, that carry out the necessary steps to enforce any policy decisions in their own control domain.

Figure 4 below illustrates the different functions of the CRI Resource Manager daemon.

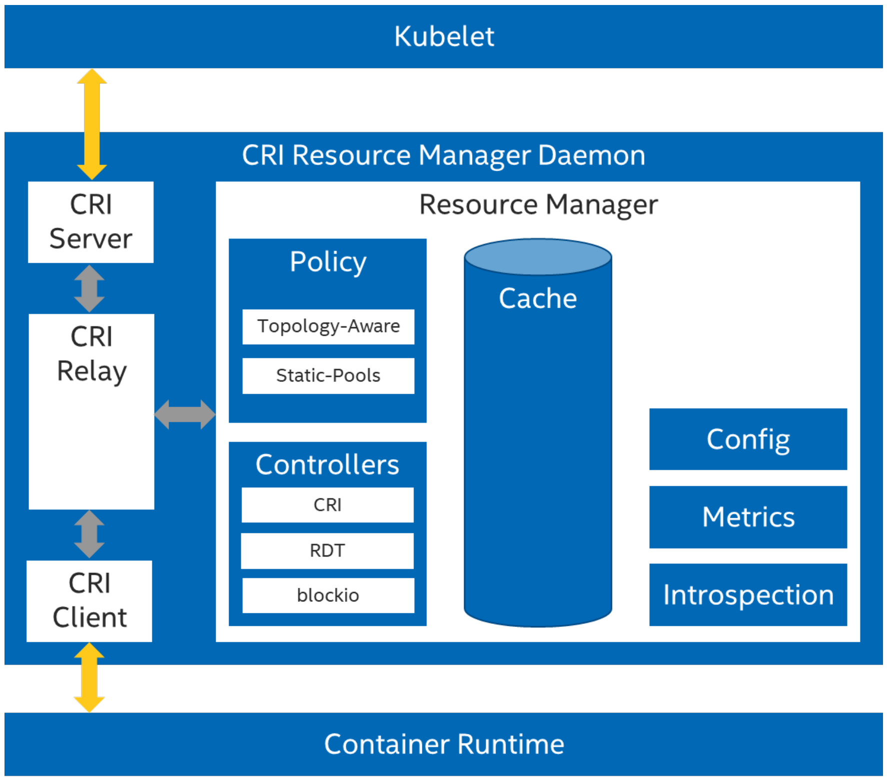

Figure 4. High-level Architecture of the CRI Resource Manager Daemon

2.3.1.1 Available Resources and Controls

CRI Resource Manager has a wide range of controls for managing workload resources. It supports all control points supported by the CRI protocol. In addition to that, CRI Resource Manager implements multiple “out-of-band” controllers that adapt to technologies and mechanisms not supported by the container runtime interface.

Table 4. Resource Controls on CRI Resource Manager

CONTROL	USER SETTABLE	NOTES
CRI Protocol		
Kernel CFS CPU parameters		
Shares (µs)	Yes	Corresponds to CPU request from Pod Spec
Quota (µs)	Yes	Corresponds to CPU limit from Pod Spec
Period (µs)	No	
Memory limit	Yes	Memory limit from Pod Spec
OOM score adjustment	Yes	Memory request from Pod Spec
Pinning to CPU set	No	
Pinning to memory controller	No	
Additional Special Controllers		
Intel® Resource Director Technology (Intel® RDT)		
RDT Class	Yes	Defaults to Pod QoS class Class configuration specifies the available LLC size and memory bandwidth
blkio cgroup controller		
blkio Class	Yes	Defaults to Pod QoS class Class configuration specifies the I/O scheduler parameters and I/O limits
Memory demotion	Limited	Dynamic monitoring of workload, periodic demotion of idle pages

2.3.1.2 Available Policies

There are currently two CRI-RM policies that are particularly interesting with respect to mitigating noisy neighbors and running latency critical workloads.

2.3.1.2.1 Topology-Aware Policy

The topology-aware policy optimizes workload placement in the node by using a detailed model of the actual hardware topology of the system. Its goal is to give measurable benefits in an automated manner with minimal input required from the user, while still enabling flexible user controls for fine-grained tuning where required.

The policy automatically detects the hardware topology of the system and, based on that, builds a fine-grained tree of resource pools from which resources are allocated to containers. The tree corresponds to the memory and CPU hierarchy of the system.

The main goal of the topology-aware policy is to distribute containers among the pools (nodes in the tree) in a way that both maximizes performance and minimizes interference between workloads. In order to achieve this, the policy implements a multitude of mechanisms that are listed in Table 5 below.

Table 5. Topology-Aware Workload Placement Optimization

TECHNOLOGY	FEATURE DESCRIPTION
CPU pinning	Pins workloads to CPU cores and memory controllers
CPU isolation	Isolates critical workloads by exclusive CPU allocation Detects kernel-isolated CPUs (isolcpus) and reserves them for critical workloads Supports mixed exclusive–shared allocation for intra-Pod prioritization
CPU prioritization	Detects CPU cores with elevated base frequency by using Intel® Speed Select Technology Base Frequency (SST-BF) Detects CPU cores with elevated priority by using Intel® Speed Select Technology Core Power (SST-CP)
Workload isolation	Isolates workloads from each other by putting them into different pools when possible
Load spreading	Assigns workload to the idlest* of best fitting pools *the exact definition of ‘idlest’ is subject to evolve with optimizations in the implementation
Resource alignment	Supports implicit ‘topology hints’ detected from hardware devices. Selects a pool closest to the assigned devices.
Memory tiering	Detects Intel® Optane™ Persistent Memory (PMEM)
Cold-start	Memory allocation of the workload can be restricted to PMEM-only for an initial warm-up period
Dynamic demotion	Idle memory pages of a workload are periodically ‘demoted’ from DRAM to PMEM
Container affinity	Supports intra- and inter-pod cross-container affinity and anti-affinity annotations
Overrides	Workload may opt out from most of the automatic CPU isolation and prioritization (reject exclusive, isolated, and/or mixed allocation)

2.3.1.2.2 Static-Pools Policy

The static-pools built-in policy was inspired by CMK (CPU Manager for Kubernetes) and is a solution for refined CPU pinning and isolation. The static-pools policy is centered around the concept of predefined pools of CPU cores. The system is partitioned into a set of non-overlapping CPU pools from which CPU cores are allocated according to the user input specified in the workload spec. It uses the construct of exclusive and shared pools, which provides CPU isolation for high priority workloads.

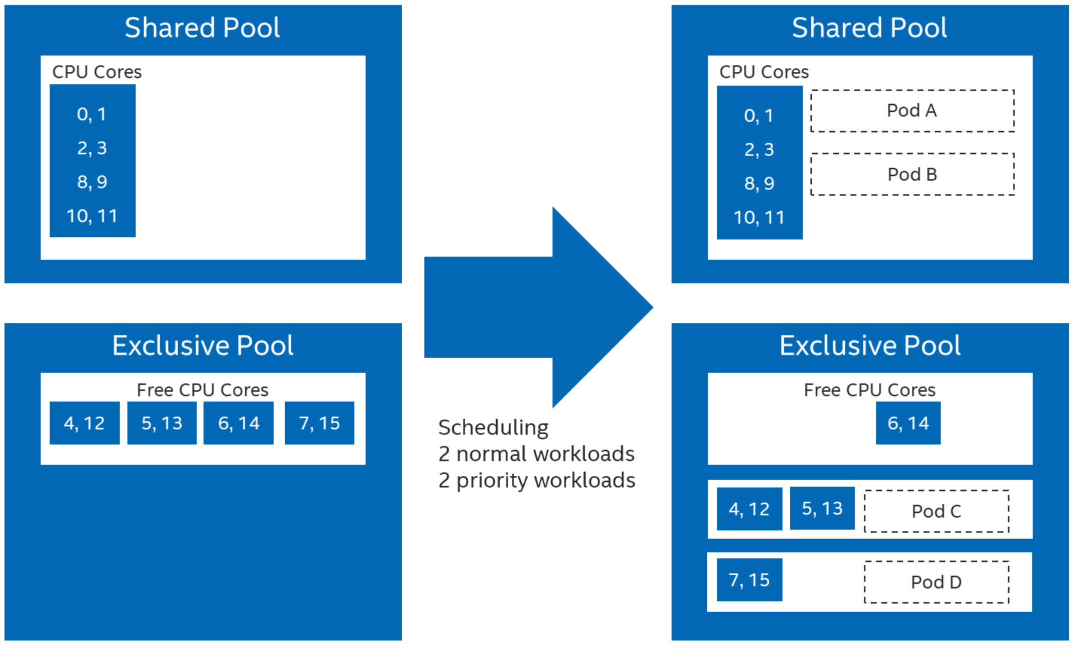

Figure 5. Concept of CPU Pools in the Static-Pools Policy

The goal of the static-pools policy is to replicate the functionality of the `cmk isolate` command of CMK. It has compatibility features to function as a drop-in replacement of CMK. It supports an arbitrary number of CPU pools and dynamic configuration via the CRI Resource Manager configuration management system. The static-pools policy can read the pools configuration directory of CMK or alternatively from the dynamic cluster-wide configuration of CRI Resource Manager. CMK compatibility features are listed in Table 6.

Table 6. CMK Compatibility Features of the Static-Pools Policy

FEATURE	SUPPORT
Legacy pools configuration from /etc/cmk	Yes
Node tainting	Yes (dynamically configurable)
Node labeling	Yes (dynamically configurable)

Parsing of `cmk isolate` command line	Yes
	`--pool`
	`--socket-id`
	`--no-affinity`

Container environment variables	Yes
	`CMK_CPUS_ASSIGNED`
	`CMK_CPUS_INFRA`
	`CMK_CPUS_SHARED`

For more information about CMK and its configuration, see https://github.com/intel/CPU-Manager-for-Kubernetes.

For user documentation about the static-pools policy, see https://intel.github.io/cri-resource-manager/stable/docs/policy/static-pools.html.

2.3.1.3 CRI Resource Manager Node Agent

CRI Resource Manager has a separate node agent daemon that handles all communication towards the Kubernetes control plane. It isolates the resource management daemon from the control plane in terms of functionality and security. The node agent is responsible for

- handling the dynamic configuration management of CRI Resource Manager by watching specific ConfigMap objects and sending configuration updates to the CRI Resource Manager daemon
- managing the node object CRI-RM is running on (resource capacity, labels, annotations, taints)
- handling dynamic adjustments of container resource assignments

While the CRI Resource Manager daemon can run without the accompanying node agent, this always prevents the usage of centralized dynamic configuration and limits the capabilities of some policies.

2.3.1.4 CRI Resource Manager Webhook

Because of the inherent limitations of the CRI protocol, CRI Resource Manager is not able to reliably see all of the container resource requirements of the workload. CRI Resource Manager Webhook is an additional component, configured as a Mutating Admission Webhook in the Kubernetes cluster, that takes care of annotating all workloads (pods, more specifically) so that the CRI Resource Manager has an exact view of their resource requirements.

CRI Resource Manager daemon is able to run without the help of the Webhook but the accuracy of some policy decisions may be degraded. The most severe limiting consequence of running without the Webhook in place is that CRI-RM loses all visibility to Kubernetes extended resources. Therefore, any policy that relies on the use of policy-specific extended resources requires in practice the Webhook. A less significant consequence of the absence of the Webhook is that CPU requests larger than 256 full CPUs are clamped to 256 CPUs. This is extremely seldom a practical limitation, if ever. A final consequence of less significance is that without the Webhook, CRI-RM is unable to accurately determine the memory requests of Burstable containers.

2.3.2 Topology-Aware Resource Alignment

The topology-aware policy automatically builds a tree of pools based on the detected hardware topology. The pool nodes at various depths from bottom to top represent NUMA nodes, dies, sockets, and finally the whole of the available hardware at the root.

As their resources, the NUMA leaves among the pool nodes are assigned the memory behind their controller and the CPU cores with the smallest distance to the controller. Each pool node upper in the tree has as resources the union of the resources in their children. If the machine has both PMEM and DRAM and is running in two-layer memory mode, the resources from CPU-less PMEM-only NUMA nodes are assigned to the closest NUMA node (or to one of the closest if this is not unique).

With this setup, each pool in the tree has topology-aligned CPU and memory resources. The amount of available resources, and the penalty of memory access and data transfer between the CPU cores, gradually increases in each pool node from bottom to top. Moreover, the resource sets of sibling pool nodes at the same depth in the pool tree are disjoint, while descendant pools overlap, while pools on the same path between any node and the root have overlapping resources.

With this setup, the topology-aware policy should handle the topology-aware alignment of resources without any special or extra configuration. When allocating resources, the policy filters out all pools with insufficient free capacity, then runs a scoring algorithm for the remaining ones, picks the one with the best score, and assigns resources to the workload from there. Although the details of the scoring algorithm are subject to change as the policy implementation evolves, its basic principles are as follows.

- It prefers pools that are lower in the tree. In other words, it prefers tighter alignment and lower latency.
- It prefers idle pools over busy ones. In other words, it prefers more remaining free capacity and fewer workloads.

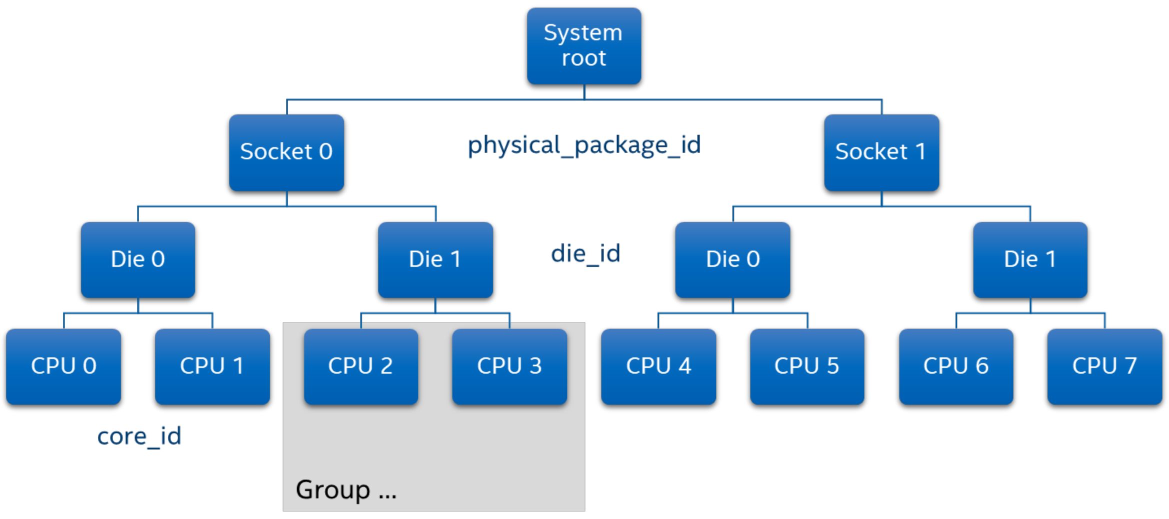

Figure 6. System CPU Hierarchy

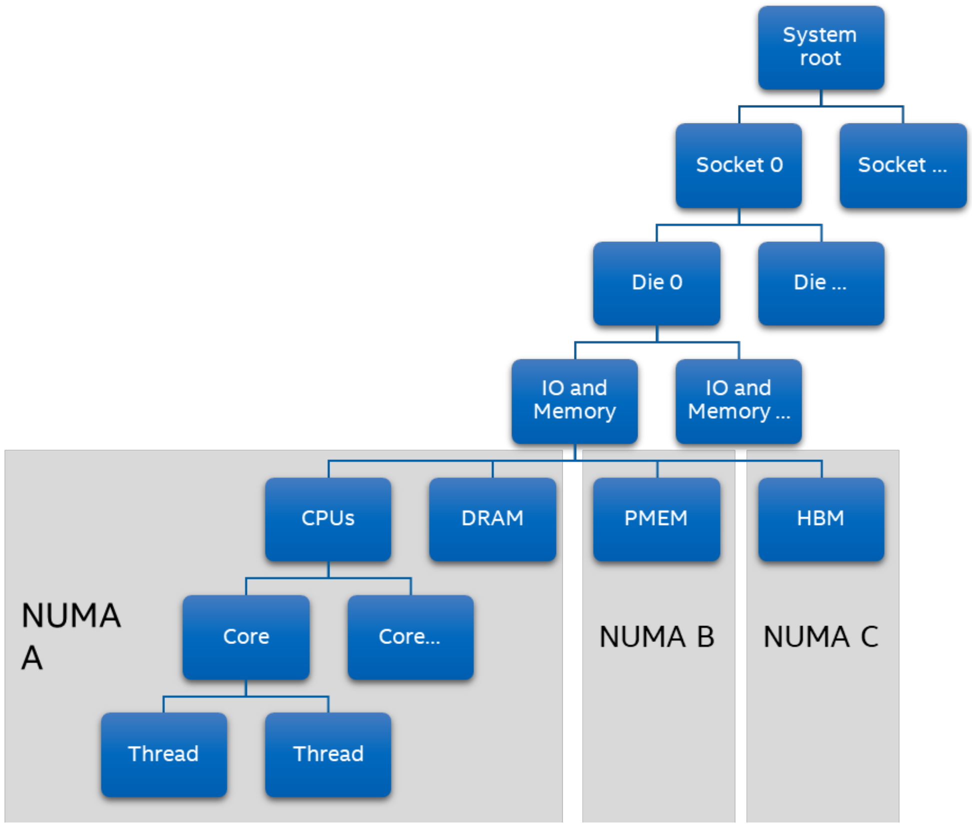

Figure 7. Representation of the System Topology Inside Topology-Aware Policy

2.3.3 Noisy Neighbors

The workload placement algorithm of the topology-aware policy tries to fit workloads as close to leaf nodes in the pool tree as possible, raising them upper in the tree only if other, better aligned placements are impossible. It also tries to spread out workloads assigned to pools at the same depth in the tree as evenly as possible over the available pools at that depth. Consequently, the policy tries to provide both the best possible alignment and the best possible isolation for workloads.

The topology-aware policy never misaligns resource allocation per se. Instead, if a workload cannot be put in a pool with optimal resource alignment, requirements for alignment are gradually softened until a suitable pool for the workload is found. This might happen due to the high overall utilization of the cluster node, or it can happen due to the sheer size of the workload itself, for example if it requests more memory or CPU cores than available in any single pool leaf node.

The topology-aware policy always assigns non-critical workloads to run on the shared subset of CPU cores within a pool. These cores consist of those non-isolated cores that are assigned to run any critical workload. The CPU cores allocated to critical workloads are always isolated from the rest by exclusive allocation. The strictness of isolation for memory access and last level cache can gradually deteriorate if an increasing number of workloads do not fit into leaf nodes, for example due to growing cluster node utilization. The policy then needs to start lifting some workloads to upper pool nodes in the tree, with resources partially overlapping with those of the child pool nodes.

In cases where the built-in default heuristics turn out to be insufficient, additional isolation can be imposed on noisy neighbors. CRI Resource Manager provides additional control over last level cache and memory bandwidth using Intel® Resource Director Technology (Intel® RDT) and the RDT controller. You can limit how much of these shared resources non-critical workloads can access by configuring corresponding RDT classes for the three Kubernetes QoS classes: BestEffort, Burstable, and Guaranteed.

You can limit the cache and memory bandwidth of these, for example to 33%, 66%, and 100% respectively, using the following CRI Resource Manager configuration fragment.

```
# sample configuration fragment for restricting LLC access by non-critical workloads
rdt:
  options:
    l3:
      # Bail out with an error if cache control is not supported
      optional: false
    mb:
      # Ignore memory bandwidth allocation settings if MBA is not supported by the system
      optional: true
  partitions:
    shared:
      # We divide up the full 100% of the available resources into classes below
      l3Allocation: "100%"
      mbAllocation: ["100%"]
      classes:
        # - critical workloads can access the full 100% of LLC and the L2 bandwidth
        # - non-critical burstable workloads can access 66% of LLC and the L2 bandwidth
        # - best effort workloads can access 33% of LLC and the L2 bandwidth
        Guaranteed:
          l3Schema: "100%"
          mbSchema: ["100%"]
        Burstable:
          l3Schema: "66%"
          mbSchema: ["66%"]
        BestEffort:
          l3Schema: "33%"
          mbSchema: ["33%"]
```

2.3.4 Latency-Critical Workloads

2.3.4.1 Topology-Aware Policy

Even if the topology-aware policy does a good job of allocating topologically aligned resources and isolates workloads from each other, sometimes this is not enough for some of the most latency critical workloads, for example some NFV applications. In such cases there are a few more things one can, and usually needs, to do to bring the performance of these critical workloads to the required level.

- Isolate CPUs at the kernel level for the critical workloads.
- Use Intel® Speed Select Technology, if available, to boost the clock frequency of kernel-isolated cores.
- Allocate exclusive cache for the critical workloads.

When CPU cores are isolated at the kernel level, they are exempt from the normal pool of CPUs used by the kernel scheduler to run and load-balance processes on the node. This provides an additional layer of protection against interference from other workloads and system processes running on the node. CRI-RM automatically detects kernel-isolated cores and reserves them for those critical workloads in the Guaranteed QoS class that also match a few additional criteria. The criteria are that the pod either needs to be explicitly annotated to opt-in to kernel-isolated allocation, or, it needs to require a single full CPU core and must not be annotated with kernel-isolation opt-out. You can annotate a container to opt out from kernel-isolated allocation with the following CRI-RM-specific annotation in the Pod Spec YAML:

```
# sample annotation to opt out all containers in a Pod from isolated CPU allocation
metadata:
  name: $POD_NAME
  annotations:
    prefer-isolated-cpus.cri-resource-manager.intel.com/pod: false
```

If only selected containers need to opt out within the pod, you can do it using the following annotation syntax:

```
# sample annotation to opt out selected containers in a Pod from isolated CPU allocation
metadata:
  name: $POD_NAME
  annotations:
    prefer-isolated-cpus.cri-resource-manager.intel.com/container.$CONTAINER_NAME1: false
```

Opting in happens in the same way, but you set the annotation value to 'true' instead of 'false'.

If you have a large number of pods and containers that you need to opt out from kernel-isolated allocation, it is often easier to change the default preference globally at the topology-aware policy level. You can do this by including the following fragment in your CRI-RM configuration:

```
# sample configuration fragment to disable isolated allocation by default
policy:
  Active: topology-aware
  topology-aware:
    PreferIsolatedCPUs: false
```

With such a configuration in place, every eligible container needs to opt in to get isolated CPUs. There are a few additional things to remember when isolating CPU cores. First, you usually do not want to isolate CPU core #0 or its HT sibling. Core #0 is somewhat special, for example some interrupts might always be handled by core #0. Second, it is best to isolate both HT siblings in any core you choose.

Allocating exclusive cache for critical workloads can be done in a similar way as for noisy neighbors. The only difference here is that we create two 'cache partitions,' one for critical and another for non-critical workloads. In the critical partition we create a single class dedicated to critical workloads. The non-critical partition we further subdivide for the three QoS classes. Since partitions do not overlap, this results in a dedicated portion of the cache in exclusive use by critical containers. Here is the configuration fragment illustrating this. Note that the various choices for partition and class sizes are only illustrative. You need to adjust these to your workloads' needs, considering things like its working set size and other characteristics.

```
# sample configuration fragment for restricting LLC access by non-critical workloads
rdt:
  options:
    l3:
      # Bail out with an error if cache control is not supported.
      optional: false
  partitions:
    critical:
      # We dedicate 60% of the available cache to critical workloads, exclusively
      l3Allocation: "60%"
      classes:
        # - critical workloads get all of the cache lines reserved for this partition
        Critical:
          l3schema: "100%"
    noncritical:
      # Allocate 40% of all cache lines to non-critical workloads.
      l3Allocation: "40%"
      classes:
        # - critical workloads can access the full 100% of the non-critical allocation.
        # - non-critical burstable workloads can access 75% of the non-critical allocation.
        # - best effort workloads can access 50% of the non-critical allocation.
        Guaranteed:
          l3schema: "100%"
        Burstable:
          l3schema: "75%"
        BestEffort:
          l3schema: "50%"
```

Additionally, we need to annotate our critical workloads to use the correct RDT class instead of the default 'Guaranteed' one in the non-critical partition. This can be done using the following annotation, for all containers in the pod:

```
# sample annotation to assign all containers to the critical RDT class
metadata:
  name: $POD_NAME
  annotations:
    rdt-class.cri-resource-manager.intel.com/pod: critical
```

If you need to assign only selected containers in the pod to the critical RDT class, you can do it with this fragment:

```
# sample annotation to assign selected containers to the critical RDT class
metadata:
  name: $POD_NAME
  annotations:
    rdt-class.cri-resource-manager.intel.com/container.$CONTAINER_NAME1: critical
```

The topology-aware policy also supports mixed allocations. An allocation of CPU cores is said to be mixed if it includes both exclusively allocated cores and the shared subset of CPU cores in the pool. The exclusive cores can be either kernel-isolated or normal ones, depending on availability. These types of allocations are sometimes useful if a single workload, or more precisely a single container within a workload, consists of both critical and non-critical processes or threads. For example, a latency-critical process pumping data to the network at wire speeds, and another process running a control protocol related to the data pump.

By default, the topology-aware policy considers any request by Guaranteed pod containers with CPU requirements between one and two CPU cores as a request for mixed allocation. A workload can be annotated to opt out from mixed allocation with the following fragment:

```
# sample annotation to opt all Pod containers out from mixed CPU allocation
metadata:
  name: $POD_NAME
  annotations:
    prefer-shared-cpus.cri-resource-manager.intel.com/pod: true
```

Or using the same annotation but only for selected containers:

```
# sample annotation to opt selected containers out from mixed CPU allocation
metadata:
  name: $POD_NAME
  annotations:
    prefer-shared-cpus.cri-resource-manager.intel.com/container.$CONTAINER_NAME1: true
```

Again, it is possible to change the global topology-aware default and then annotate containers to opt-in to mixed allocations. That can be done with the following configuration fragment:

```
policy:
  Active: topology-aware
  topology-aware:
    PreferSharedCPUs: true
```

When mixed allocations are in use, the workload is allowed to run on all the CPUs, including both the shared and the exclusive ones. In such cases, the workload itself needs to arrange for its processes to get pinned to the right exclusive CPU cores or the shared subset. To help with this, CRI-RM exposes the set of resources allocated to a container in a plain text file with Bourne shell compatible syntax. This file is bind-mounted in the container's filesystem under the path `/.cri-resmgr/resources.sh`. Within this file, there are three keys/variable names related to CPU allocations. These are SHARED_CPUS, EXCLUSIVE_CPUS, and ISOLATED_CPUS, and they indicate the correspondingly allocated CPUs using the kernel's dash- and comma-separated CPU set notation. This file is also updated whenever the allocation for the container changes. The inotify family of system calls can be used on this file to receive change notification events when this file, and consequently the resource allocation of the container, changes.

2.3.4.2 Static-Pools Policy

Unlike the topology-aware policy that slices the system into dynamic CPU and memory pools and balances workloads automatically, the static-pools policy gives low-level control on configuring CPU cores to pools and assigning workloads into them. Functionally this is similar to what can be achieved with the CPU Manager for Kubernetes (CMK). For easy transition from CMK, the static-pools policy has backwards compatibility features that enable using it as a drop-in replacement for CMK.

If and only if a workload requests exclusive CPUs is it executed on CPUs listed in the exclusive pool. This helps with the noisy neighbor problem to the extent that the set of workloads using certain CPUs is known.

Enabling the static-pools policy and assigning workloads to CPU pools requires these steps:

1. Enable CRI Resource Manager Webhook (described in detail in Section 3.4).
2. Optionally, create pool configuration on nodes with `cmk init`.
3. Configure CRI Resource Manager policy.
4. Include pool and CPU requirements into the workload pod spec.

The pool configuration may be created by running the `cmk init` command on each of the nodes of the cluster. For detailed instructions, see CPU Management - CPU Pinning and Isolation in Kubernetes* Technology Guide. Alternatively, the pools can be specified under the static-pools policy configuration outlined below.

The snippet below describes how to activate the static-pools policy in a CRI Resource Manager configuration. The included pool configuration is for illustrative purposes only.

```
policy:
  Active: static-pools
  static-pools:
    # Do not create legacy CMK node taint or node label.
    LabelNode: false
    TaintNode: false
    # If pools is missing here static-pools reads pools configuration from /etc/cmk
    pools:
      exclusive:
        cpuLists:
        - Cpuset: 0,22
          Socket: 0
        - Cpuset: 1,23
          Socket: 0
        - Cpuset: 0,22
          Socket: 1
        - Cpuset: 1,23
          Socket: 1
        exclusive: true
      shared:
        cpuLists:
        - Cpuset: 6-21,28-43
          Socket: 0
        - Cpuset: 3-21,24-43
          Socket: 1
        exclusive: false
      infra:
        cpuLists:
```

```
        - Cpuset: 2-5,24-27
          Socket: 0
        exclusive: false
```

When the CRI Resource Manager has been configured and the CRI Resource Manager Webhook is active in the cluster, the following pod spec launches a process that gets one exclusive CPU (`cmk.intel.com/exclusive-cores`) in the exclusive pool (`STP_POOL`) that is run on CPU socket 1 (`STP_SOCKET_ID`).

```
apiVersion: v1
kind: Pod
metadata:
  name: critical
spec:
  containers:
    - name: mycont
      image: busybox
      env:
        - name: STP_POOL
          value: 'exclusive'
        - name: STP_SOCKET_ID
          value: '1'
      command: ['sh', '-c', 'sleep inf']
      resources:
        requests:
          cpu: 1000m
          cmk.intel.com/exclusive-cores: '1'
        limits:
          cpu: 1000m
          cmk.intel.com/exclusive-cores: '1'
```

For compatibility with CMK, instead of using environment variables (`STP_POOL`, `STP_SOCKET_ID`) in the last step, the `cmk isolate` command can be used as well. However, the `cmk` tool is not needed in container images. The static-pools policy parses the container command line and transparently removes `cmk isolate` and its arguments.

3 Deployment

3.1 Setting Up Cluster Nodes with CRI Resource Manager

3.1.1 Install CRI Resource Manager

CRI Resource Manager can be installed to a node from pre-built binary packages. On CentOS, Fedora, and SUSE:

```
CRIRM_VERSION=0.4.1 # You probably want to check and use the latest tagged release.
source /etc/os-release
sudo rpm -Uvh https://github.com/intel/cri-resource-
manager/releases/download/v${CRIRM_VERSION}/cri-resource-manager-${CRIRM_VERSION}-
0.x86_64.${ID}-${VERSION_ID}.rpm
```

On Ubuntu and Debian:

```
CRIRM_VERSION=0.4.1 # You probably want to check and use the latest tagged release.
source /etc/os-release
pkg=cri-resource-manager_${CRIRM_VERSION}_amd64.${ID}-${VERSION_ID}.deb
curl -LO https://github.com/intel/cri-resource-manager/releases/download/v${CRIRM_VERSION}/${pkg}
sudo dpkg -i ${pkg}; rm ${pkg}
```

3.1.2 Launch CRI Resource Manager

Packages include `systemctl` service files for CRI Resource Manager. When installed, the following uses the default configuration template as is and launches CRI Resource Manager.

```
sudo cp /etc/cri-resmgr/fallback.cfg.sample /etc/cri-resmgr/fallback.cfg
sudo systemctl enable cri-resource-manager && sudo systemctl start cri-resource-manager
```

Make sure that `cri-resmgr` connects to the container runtime that is used by kubelet. You should see the same PATH `in --runtime-socket PATH` parameter as kubelet.

3.1.3 Inject CRI Resource Manager Between Kubelet and the Container Runtime

If a cluster node is already running kubelet, you need to change its command line arguments so that it connects to CRI Resource Manager instead of a container runtime. That is, kubelet needs the following arguments:

```
--container-runtime=remote --container-runtime-endpoint=/var/run/cri-resmgr/cri-resmgr.sock
```

If a cluster node is being created, you can pass `cri-resmgr.sock` directly to the node creator. For example, with kubeadm:

```
kubeadm join --cri-socket /var/run/cri-resmgr/cri-resmgr.sock ...
```

3.2 Setting Up Cluster-Wide Configuration for CRI Resource Manager

In addition to using local configuration files per node, CRI Resource Managers running on cluster nodes can be configured via cluster-wide configuration. If a CRI Resource Manager node agent is deployed on a node, the agent monitors changes in specific-to-node, specific-to-group, and the cluster default ConfigMaps, named `cri-resmgr-config.node.NODE_NAME`, `cri-resmgr-config.group.GROUP_NAME`, and `cri-resmgr-config.default`, respectively. If the agent observes changes in matching ConfigMaps, it passes the new configuration to the CRI Resource Manager running on the node.

The data section of ConfigMaps contains the same information as a local configuration file. For example, a default ConfigMap that defines block I/O read throttling and lower block I/O priority on pods with annotation `blockioclass.cri-resource-manager.intel.com/pod: ThrottledIO` looks like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cri-resmgr-config.default
  namespace: kube-system
data:
  policy: |+
    Active: memtier
    ReservedResources:
      CPU: 750m
  blockio: |+
    Classes:
      ThrottledIO:
        - Weight: 80 # The default is 100
        - Devices:
            - /dev/vda
          ThrottleReadBps: 100M
```

You can create a set of example ConfigMaps in the kube-system namespace with the following code.

```
CRIRM_VERSION=0.4.1
kubectl apply -f \
https://raw.githubusercontent.com/intel/cri-resource-manager/$CRIRM_VERSION/sample-configs/cri-resmgr-configmap.example.yaml
```

3.3 Setting Up CRI Resource Manager Node Agent

The CRI Resource Manager node agent is needed to enable centralized and dynamic CRI Resource Manager configuration (ConfigMaps). Also, the static-pools policy requires the node agent for managing node labels, taints, and exclusive resources.

The CRI Resource Manager node agent can be deployed on nodes as a Kubernetes DaemonSet so that it will be run on every schedulable node.

The following snippet demonstrates how to deploy CRI Resource Manager node agent as a K8s DaemonSet in the kube-system namespace:

```
CRIRM_VERSION=0.4.1
curl https://raw.githubusercontent.com/intel/cri-resource-manager/$CRIRM_VERSION/cmd/cri-resmgr-agent/agent-deployment.yaml | \
sed s",IMAGE_PLACEHOLDER,docker.io/intel/cri-resmgr-agent:$CRIRM_VERSION," | kubectl apply -f -
```

3.4 Setting Up CRI Resource Manager Webhook

The CRI Resource Manager Webhook is required by the static-pools policy. The Webhook runs as a part of the Kubernetes control plane and ensures that all resource request information in pod specifications, including extended resources, reaches the CRI Resource Manager that will launch the pod. Otherwise, part of this information is lost due to limitations in the CRI protocol.

The CRI Resource Manager Webhook must be deployed with signed certificates that make it a trusted part of the Kubernetes control plane. In addition to deploying the Webhook container, a MutatingWebhookConfiguration that hooks the Webhook as part of the control plane must be created.

Signed certificates can be created with OpenSSL, for example. The certificates are saved into the cluster inside Kubernetes Secret named `cri-resmgr-webhook-secret`. The following snippet demonstrates how to create a self-signed certificate using OpenSSL and deploy that to the cluster.

```
SVC=cri-resmgr-webhook NS=cri-resmgr
openssl req -x509 -newkey rsa:2048 -sha256 -days 365 -nodes \
  -keyout cmd/cri-resmgr-webhook/server-key.pem \
  -out cmd/cri-resmgr-webhook/server-crt.pem \
  -subj "/CN=$SVC.$NS.svc" \
  -addext "subjectAltName=DNS:$SVC,DNS:$SVC.$NS,DNS:$SVC.$NS.svc"
cat >cmd/cri-resmgr-webhook/webhook-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: cri-resmgr-webhook-secret
  namespace: $NS
data:
  svc.crt: $(base64 -w0 < cmd/cri-resmgr-webhook/server-crt.pem)
  svc.key: $(base64 -w0 < cmd/cri-resmgr-webhook/server-key.pem)
type: Opaque
EOF
kubectl create namespace $NS
kubectl create -f cmd/cri-resmgr-webhook/webhook-secret.yaml
```

The CRI Resource Manager Webhook container can be deployed to the `cri-resmgr` namespace using:

```
CRIRM_VERSION=0.4.1
curl https://raw.githubusercontent.com/intel/cri-resource-manager/$CRIRM_VERSION/cmd/cri-
resmgr-webhook/webhook-deployment.yaml | \
sed s",IMAGE_PLACEHOLDER,docker.io/intel/cri-resmgr-webhook:$CRIRM_VERSION," | kubectl apply -f
-
```

Finally, the admission Webhook can be activated with:

```
curl https://raw.githubusercontent.com/intel/cri-resource-manager/$CRIRM_VERSION/cmd/cri-
resmgr-webhook/mutating-webhook-config.yaml |\
    sed -e "s/CA_BUNDLE_PLACEHOLDER/$(base64 -w0 < cmd/cri-resmgr-webhook/server-crt.pem)/" | \
    kubectl apply -f -
```

For details, see the CRI Resource Manager Webhook documentation.

4 Implementation Example

4.1 Co-locating Client and Server on a Single NUMA Zone

This example demonstrates configuring two pods so that they will be placed on the same NUMA zone. When pods communicate a lot with each other, like a database and its client in this example, this configuration minimizes communication distance in terms of interconnect hops. For the same reason, the configuration mitigates the interference to the client-server communication caused by memory traffic of other workloads running on the system. The Benefits section below shows how remarkable the latency improvement can be[1].

First, let's define a completely normal Redis database deployment. `redis.yaml` below has nothing special for workload placement.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis
          ports:
            - containerPort: 6379
              name: redis
          env:
            - name: MASTER
              value: 'true'
```

[1] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

Next, we use `memtier-benchmark` as a client for the database. `memtier-benchmark.yaml` below defines the benchmark pod's *affinity* for the `redis` server pod. The affinity tells CRI Resource Manager that a pod should be running close to another pod. The affinity section highlighted below defines affinity of weight 10 to containers named `redis` inside any pod with a name that matches wildcard `redis-*`.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: memtier-benchmark
spec:
  template:
    metadata:
      annotations:
        cri-resource-manager.intel.com/affinity: |+
          memtier-benchmark:
            - scope:
                key: pod/name
                operator: Matches
                values:
                  - redis-*
              match:
                key: name
                operator: Equals
                values:
                  - redis
              weight: 10
    spec:
      containers:
        - name: memtier-benchmark
          image: redislabs/memtier_benchmark:edge
          args: ['--server=redis-service']
      restartPolicy: Never
```

CRI Resource Manager supports anti-affinities, too. Opposite to the `affinity` annotation, `anti-affinity` defines that running a pod on the same NUMA zone with another pod should be avoided. CRI Resource Manager takes affinities and anti-affinities into account in placing workloads of any QoS class. In this example, the performance difference between anti-affinity and affinity is roughly 50 % more throughput and 35 % lower median latency for affinity, when there is nothing else but Redis and the benchmark running on the node[2].

5 Benefits

The greatest performance benefits from CRI Resource Manager are gained when a cluster node is running many memory-intensive background workloads at the same time. However, the performance is at least break even or, more often, improved even if no other workloads run on the same node, as well as in the case of CPU-intensive workloads[3].

On December 16 through December 18, 2020, we benchmarked a Redis in-memory database with the `memtier-benchmark` performance test. The test reports median, 99 %, and 99.9 % percentile latencies in database GET/SET operations. The lower the latency the better the performance. The benchmark environment is described in the table below.

Table 7. Benchmark Environment

FEATURE	DESCRIPTION
CPU	56 cores (112 hyper-threads), Intel® Xeon® Platinum 8280L CPU @ 2.70GHz
Memory	128 GB DRAM + 512 GB PMEM - 4 NUMA zones with 28 CPUs and 32 GB of DRAM - 4 NUMA zones with no CPU and 128 GB of PMEM
OS	Linux 5.10.0-rc6-2 openSUSE Leap 15.2
Cloud	Kubernetes v1.19.3, single node cluster CRI Resource Manager 0.4.0-275 with the default configuration containerd v1.2.13

[2] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

[3] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

FEATURE	DESCRIPTION
Background workloads	stress-ng 0.10.10 running CPU- and memory-intensive workers 256 workers (16 workers per container, 8 containers per pod, 2 pods)
Benchmark workloads	Redis 6.0.9 in-memory database Redislabs/memtier-benchmark 1.3.0

5.1 Benefits on Critical Workloads

In this configuration, background workloads are memory intensive (stress-ng memcpy) and belong to the Burstable QoS class. Benchmark workloads belong to the Guaranteed QoS class, that is, they are considered more important than background workloads.

Table 8. Background Workloads Belong to Burstable QoS Class and Benchmark Workloads Belong to Guaranteed QoS Class

REDIS LATENCY ON GUARANTEED BENCHMARK	MEDIAN	99 % PERCENTILE	99.9 % PERCENTILE
kubelet defaults CRI-RM + containerd	1.0 x	2.5 x	4.2 x
kubelet with CPU + topology managers containerd	1.0 x	9.0 x	15 x
kubelet defaults containerd	11 x	23 x	39 x

Note that CRI Resource Manager results are measured with the default configuration that is included in the CRI-RM. It is possible to isolate critical workloads even better, with the RDT for example, to get even lower and more predictable latencies.

5.2 Benefits on Overall Performance

In the following configurations, background workloads are memory intensive (stress-ng memcpy). In the first table both background and benchmark workloads belong to the Burstable QoS class, which means that their performance is optimized similarly. This demonstrates overall performance gain. Note that latency factors are comparable only within each table. Latencies in Guaranteed, Burstable, and BestEffort cases are very different, as shown in Section 5.3.

Table 9. Background and Benchmark Workloads Belong to Burstable QoS Class

REDIS LATENCY ON BURSTABLE BENCHMARK[4]	MEDIAN	99 % PERCENTILE	99.9 % PERCENTILE
kubelet defaults CRI-RM + containerd	1.0 x	2.1 x	3.4 x
kubelet containerd	1.5 x	3.0 x	4.2 x

Finally, a comparison where all background and benchmark workloads belong to the BestEffort QoS class. That is, no CPU or memory resource requirements whatsoever are specified for any of the pods.

Table 10. Background and Benchmark Workloads Belong to BestEffort QoS Class

REDIS LATENCY ON BESTEFFORT BENCHMARK	MEDIAN	99 % PERCENTILE	99.9 % PERCENTILE
kubelet defaults CRI-RM + containerd	1.0 x	3.8 x	7.3 x
kubelet containerd	1.7 x	5.0 x	16 x

Here we do not report separately kubelet CPU and topology managers from kubelet default parameters because CPU and topology manager have no effect on Burstable or BestEffort pods.

4 See backup for workloads and configurations or visit http://www.Intel.com/PerformanceIndex. Results may vary.

5.3 Benefits of Higher QoS class

The previous results show how CRI Resource Manager cuts latencies and makes them far more predictable compared to running without CRI-RM in all QoS classes. Finally, we present how big a difference the QoS class of benchmark pods makes to latencies. The following table compares only the best cases above, that is, running with CRI Resource Manager.[5]

Table 11. Benchmark Latency on Different QoS Classes Running with CRI Resource Manager

REDIS LATENCY ON DIFFERENT QOS CLASSES	MEDIAN	99 % PERCENTILE	99.9 % PERCENTILE
Benchmark Guaranteed Background Burstable	1.0 x	2.5 x	4.2 x
Benchmark Burstable Background Burstable	10 x	21 x	34 x
Benchmark BestEffort Background BestEffort	22 x	82 x	160 x

6 Summary

CRI Resource Manager helps mitigate the noisy neighbor problem and optimizes overall workload performance on Intel processors. It offers both configurable and fully automatic measures towards these ends. Configuration enables dedicating CPU cache, memory, and block I/O bandwidth to workloads. Automation takes care of aligning and pinning workloads to memory zones, CPUs, and other node resources so that communication delays and interference are minimized. The higher the system load, the bigger the performance benefit from using the CRI Resource Manager, even when running it as delivered with the default configuration.

[5] See backup for workloads and configurations or visit http://www.Intel.com/PerformanceIndex. Results may vary.

0421/DN/WIPRO/PDF 635703-001US