OpenShift Virtualization - Tuning & Scaling Guide

Updated November 8 2023 at 2:56 PM - English

Table of Contents

Overview

Goal

OpenShift Container Platform defaults are designed to work well out of the box. Many pod and VM scenarios and recommendations are documented in Scalability and Performance. In addition, virtual machine (VM) templates, by default, include a High-performance workload type.

This tuning guide is a supplemental document. It focuses on fine-tuning cluster scalability and VM performance for several use cases and environments, for a variety of workloads and cluster sizes. This document provides guidance for scaling up and scaling out an environment, up to 100 OpenShift nodes based on this reference architecture, and for highly tuning each component of a VM definition.

> **Note**
> Determining scaling recommendations and limitations is heavily dependent on the exact use case. The impact of scaling each cluster dimension should be evaluated against workload performance needs. Large cluster sizes reaching beyond 100 - 150 nodes might be better managed as separate control planes using Red Hat Advanced Cluster Management.

Guide: Cluster configurations

In general, there are a few configuration details that should be considered for larger scale clusters, regardless of whether workloads run as pods or VMs. This section summarizes some key areas for general cluster performance.

Etcd

It is very important to use high-speed storage for master nodes running etcd, especially for larger clusters. We recommend at least SSD (preferably NVMe)-backed local storage.

Your hardware should be capable of providing storage performance such that etcd write-ahead-log fsync duration and network peer round trip time complete within the recommended thresholds. See the following guides on etcd performance:

- Recommended etcd practices
- Backend performance requirements for OpenShift etcd
- How to use 'fio' to check etcd disk performance in OCP

Infrastructure components

Restricting critical cluster components to “infra” nodes allows them to properly grow along with cluster/workload scalability. It also provides isolated resources to prevent disruption of worker node performance. See Infrastructure Nodes in OpenShift 4 for configuring components such as the router, registry, monitoring, and logging.

Monitoring

In addition to “infra” node isolation, consider configuring persistent storage for Prometheus and AlertManager.

This provides two advantages:
1. Metric and Alert data will survive node/pod restarts, up to the configured retention time.
2. PromDB will not fill up the RHCOS sysroot partition, which can cause a node to become unavailable.

See Configuring the monitoring stack in the OpenShift documentation for details.

Note that PromDB growth rates, especially in terms of memory usage and storage size, depend on many factors including pod churn rates, VM start/stop rates, number of VM migrations, number of PVCs, and configured retention time.

Tested maximums

Consider the following tested object maximums when running VMs on OpenShift. These values are based on the largest possible cluster size and reaching near the maximum values may reduce performance and increase response latency, carefully consider all of the multidimensional factors that limit the cluster scale.

> Note: These guidelines apply to OpenShift Container Platform with software-defined networking (SDN), not Open Virtual Network (OVN).

These maximums and minimums apply to OpenShift Virtualization 4.x as a large-scale environment. The limits apply on a per-cluster basis. Keep in mind that some applications may work well in an overcommitted CPU environment. However, currently memory cannot be overcommitted for VMs.

> Note: The correct values depend on the use-case, i.e. it's not necessarily helpful to always trend towards the maximums, instead multiple smaller clusters can be used instead.

VM maximums

The following maximums apply to the VMs running on OpenShift Virtualization:

Objective	4.x tested maximum	Theoretical limit
Maximum concurrently running virtual machines	10,000	
Maximum virtual CPUs per virtual machine	216	384[1]
Maximum memory per virtual machine	6 TB[1]	
Minimum memory per virtual machine	N/A	
Maximum single disk size per virtual machine	2 TB	

Host maximums

The following maximums apply to the OpenShift hosts used for OpenShift Virtualization:

Objective	4.x tested maximum	Theoretical limit
Logical CPU cores or threads	Same as RHEL	
RAM	Same as RHEL	
Simultaneous live migrations	Default to 2 outbound migrations per node, and 5 concurrent migrations per cluster	Depends on NIC bandwidth
Live migration bandwidth	There is no default bandwidth limit for live migrations	Depends on NIC bandwidth

Cluster maximums

The following maximums apply to objects defined in OpenShift Virtualization:

Objective	4.x tested maximum	Theoretical limit
Maximum number of attached PVs per node		CSI storage provider dependent
Maximum PV size		CSI storage provider dependent
Maximum number of hosts	500 (<100 recommended)[2]	Same as OpenShift
Maximum number of VMs	10,000[3]	Same as OpenShift

Worker MCP configuration

If you have a large number of nodes and want to reduce serial reboot times, consider increasing the maxUnavailable setting in the worker MachineConfigPool to speed up roll-outs of MachineConfig changes and cluster updates, this setting will control how many workers can be drained in parallel. Note that if multiple worker MCPs are created, by default all pools can drain in parallel, otherwise the pool draining behavior can be controlled by pausing or unpausing each pool.

Some considerations when increasing the number of unavailable nodes:

If any installed operators or deployments have set a `PodDisruptionBudget` , it is important to have enough available nodes to meet that budget. To list all cluster PDBs, run the following command:

```
$ oc get pdb -A
```

If nodes are providing OpenShift Data Foundation storage, MCPs can be created to match the data replica topology to ensure the PDB allows parallel node drains for a single pool, by first labeling nodes with the 3 “rack” topology labels before the StorageCluster install:

```
$ oc label node {$nodeA} topology.rook.io/rack=rack0
$ oc label node {$nodeB} topology.rook.io/rack=rack1
$ oc label node {$nodeC} topology.rook.io/rack=rack2
```

Then you create 3 node roles and pools to match the 3 rack topology labels.

If the cluster is running migratable VMs, the migration resulting from any node drains will require adequate memory (and cpu) request capacity to allow a new destination virt-launcher pod to be scheduled on a new worker node. Confirm that the combined available node spare request capacity is sufficient to account for the `(total number of nodes draining in parallel) x (total number of VMs allowed to migrate in parallel, based on the cluster migration limits) x (VM mem/cpu request)`. See Configuring live migration limits and timeouts in the OpenShift documentation.

Scheduling

By default Kubelet reports a list of container images residing on each node, sorted in order of largest to smallest, up to the `nodeStatusMaxImage` count. This limit was created to ensure API objects do not become very large. The image list is used to determine a scheduling score for the `imageLocality` plugin, which OpenShift uses as part of the overall default scheduling score.

To ensure balanced node scheduling on clusters that can potentially have many container images, it is recommended to disable the `nodeStatusMaxImage` count so that the `imageLocality` scheduler plugin does not “override” the `nodeResource` scheduler scores because there are more than the default number of container images on a node (50). See the note in Managing nodes for details. To do so, apply a KubeletConfig to disable the `nodeStatusMaxImages` value.

Label worker nodes for the KubeletConfig change:

```
$ oc label machineconfigpool worker custom-kubelet=max-node-images
```

Apply the KubeletConfig change:

Note: this will reboot all workers to apply a newly generated MachineConfig.

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: max-node-images
spec:
  machineConfigPoolSelector:
      matchLabels:
         custom-kubelet: max-node-images
  kubeletConfig:
      nodeStatusMaxImages: -1
```

Max pod limit

When calculating total desired object density, consider that the default maxPod limit of 250 pods per node includes cluster and operator pods. It is possible to raise the maxPods value, but keep in mind increasing pod density can lead to additional control plane stress.

Also plan for the fact that VM migration will temporarily create an additional pod during the migration process, factor this into density planning based on how many total VMs will be allowed to migrate at once.

KubeAPI Burst/QPS

Increasing the default kubeAPI Burst and Queries Per Second (QPS) values can improve performance of bulk object creation at scale.

By default the values are set to 100 and 50 respectively to keep API server compute resource utilization reasonably low to accommodate small nodes, however when scaling to high total pod counts it can help to double the default values using the KubeletConfig.

Label worker nodes for the KubeletConfig change:

```
$ oc label machineconfigpool worker custom-kubelet=set-api-rates
```

Apply the KubeletConfig change:

Note: this will reboot all workers to apply a newly generated MachineConfig.

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-api-rates
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: set-api-rates
  kubeletConfig:
    kubeAPIBurst: 200
    kubeAPIQPS: 100
```

Guide: VM tuning

Virt control plane tuning

Some tuning options are available at the Virtualization control plane level, including increasing “burst” rates to allow creation of hundreds of VMs in a single batch and adjusting the migration settings based on the workload type.

Virt rate limiting

To compensate for large scale “burst” rates, scale up the QPS (Queries per Second) and Burst RateLimits for the virt client components (api, controller, handler, and webhook). This allows more client requests or API calls to be processed concurrently for each component preventing slow VM creation time, as can be seen by checking the ratelimiter metric (`rest_client_rate_limiter_duration_seconds_bucket`).

To apply the scalable tuning parameters recommended for large scale "High Burst" scenarios (i.e. creating many VMs at once or in large batches), enable this profile which applies tested and safe QPS and burst values for all Virt components:

> **Note:** It is recommended to open a **Support Exception** *before applying this method of tuning*

```
$ oc patch -n openshift-cnv hco kubevirt-hyperconverged --type=json -p='[{"op":
"add", "path": "/spec/tuningPolicy", "value": "highBurst"}]'
```

VM migration tuning

Live migration allows a running Virtual Machine Instance (VMI) to move to another node without interrupting the workload. Migration can be helpful for a smooth transition during cluster upgrades or any time a node needs to be drained for maintenance or configuration changes.

Live migration requires the use of a shared storage solution that provides ReadWriteMany (RWX) access mode, and that VM disks are backed by volumes defined as such. OpenShift Virtualization will check that a VMI is live migratable and if so the `evictionStrategy` will be set to LiveMigrate. See About live migration for details.

Migration limits

Cluster-wide live migration limit settings can be adjusted based on the type of workload and migration scenario to control how many VMs migrate in parallel, how much network bandwidth can be used by each migration, and how long the migration will be attempted.

If multiple VMs are migrating in parallel per node it is recommended to set the bandwidthPerMigration limit since that can prevent a large or busy VM from using a large portion of the node's network bandwidth. Note that by default the value is "0", or unlimited.

If you have a very large VM running a heavy workload (for example database processing), consider adjusting the timeout and bandwidth to allow that VM to complete migration:
1. Enable post-copy mode: `allowPostCopy: "true"` .
2. Lower `completionTimeoutPerGiB` to trigger postCopy mode sooner.
3. Increase `bandwidthPerMigration` if more throughput can be dedicated per VM (or use default of "unlimited").
4. Potentially increase `progressTimeout` for very large memory sizes running a heavy load.

> Note: When post-copy mode kicks in during a migration (after the initial pre-copy phase does not complete during the configured timeouts), the VM CPUs are paused on the source host to transfer the minimum required memory pages, then the VM CPUs are activated on the destination host and the remaining memory pages are faulted into the destination node at runtime which may cause some performance impact during the transfer. Post-copy mode is not recommended for critical data or unstable networks.

Migration network

By default, all VM migration network transfers occur over the cluster pod network, which will incur some overhead costs as the traffic is encrypted using Transport Layer Security (TLS). When the environment and application allows, using a dedicated additional network for VM migration can significantly improve network and migration performance. See Configuring a dedicated network for live migration for details.

VM host configuration tuning

Pinning

There are multiple pinning-related operators available at the host level: CPU Manager, Topology Manager, and Memory Manager.

CPU Manager

Enabling CPU Manager with a “static” policy on nodes allows VMs that request dedicated CPUs to be automatically pinned using a cpuset, it will also remove those CPUs from the shared pool ensuring that they are not shared with other VMs/pods.

CPU Manager can be enabled through a KubeletConfig. See Using CPU Manager and Topology Manager for instructions.

reservedSystemCPUs

Optionally, when configuring CPU Manager, the `reservedSystemCPUs` setting can be added to reserve specific CPUs for system work (OS and Kubernetes daemons), which can also be used in conjunction with CPU partitioning style tuning if needed for very low latency requirements.

Example KubeletConfig to apply CPU Manager along with the `reservedSystemCPUs` setting:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: cpumanager-enabled
spec:
  machineConfigPoolSelector:
    matchLabels:
    custom-kubelet: cpumanager-enabled
  kubeletConfig:
    cpuManagerPolicy: static
    cpuManagerReconcilePeriod: 5s
    reservedSystemCPUs: "0,1"
```

Topology Manager

Topology Manager allows for further control of the pinning behavior at the node level by using “hint providers” (examples: SRIOV, GPU operators) to provide preferred NUMA node information to CPU Manager when the cpuset is selected, based on the topology policy that is configured.

Topology Manager can also be configured using a KubeletConfig; see Using CPU Manager and Topology Manager for details.

Memory Manager

When VMs are small enough to fit within a NUMA node, Memory Manager can be used in conjunction with the Topology Manager policies of `single-numa-node` or `restricted` to ensure memory is also affinitized to a preferred NUMA node. See the NUMA-aware scheduling section for more information.

If a VM is large enough to span multiple NUMA nodes on a system, the memory affinitization can be controlled through a VM option called guestMappingPassthrough instead, see the [VM Tuning]{#VMTuning} section for more information.

Memory Manager can be enabled by configuring the “static” policy using a kubeletConfig similar to the CPU and Topology Managers. Configuring Memory Manager also requires setting a `reservedMemory` value for at least one NUMA node. This value should equal the sum of all reserved memory for a node `(kube-reserved + system-reserved + eviction-threshold)` . This value can be calculated by comparing a node’s total reserved value: Memory Capacity - Memory Allocatable. Example kubeletConfig syntax is provided below:

```
kubeletConfig:
  cpuManagerPolicy: static
  cpuManagerReconcilePeriod: 5s
     topologyManagerPolicy: single-numa-node
     memoryManagerPolicy: Static
     reservedMemory:
       - numaNode: 0
         limits:
           memory: “1124Mi”
```

Node Tuning Operator

The Node Tuning Operator (NTO) helps you manage node-level tuning by orchestrating the TuneD daemon and achieves low latency performance by using the Performance Profile controller. The majority of high-performance applications require some level of kernel tuning. The Node Tuning Operator provides a unified management interface to users of node-level sysctls and more flexibility to add custom tuning specified by user needs. Learn more about how to tune the host for performance using the Node Tuning Operator.

VM configuration tuning

Customizing a VM template

VM templates for multiple different operating systems and flavors are provided by default, and are easily customizable. When creating a VM from a template, it is also possible to customize the VM by selecting a Workload Type, which includes a high-performance option that uses many of the tuning options mentioned in this section. The high-performance option exchanges the compatibility of the server and desktop workload types to provide better performance at the cost of requiring CPU Manager and virtio drivers in the guest.

High-performance workload type

The default VM templates provide a “high-performance” workload type option that can be used when certain conditions apply. The additional tuning options used when this workload type is selected are summarized as follows, each option is described in more details in later sections:

For all OS types:

* `bus: virtio`
* `dedicatedCpuPlacement: true` [CPUManager “static” policy is required]
* `isolateEmulatorThread: true` [CPUManager “static” policy is required]
* `networkInterfaceMultiqueue: true`

For non-Windows OS types:

* `ioThreadsPolicy: shared`
* `dedicatedIOThread: true`

Windows guest tuning

> Note the default high-performance VM templates include some critical options for Windows performance, those options are highlighted here for informational purposes.

Hyper-V enlightenments

These Windows-specific settings instruct QEMU to use paravirtualized options that significantly improve performance, especially due to defining the best performing timer options.

The default Windows VM templates include all Hyper-V enlightenment features that are continuously tested and supported. See the example list below:

```
  spec:
    domain:
      clock:
        timer:
         hpet:
           present: false
         hyperv: {}
         pit:
           tickPolicy: delay
         rtc:
           tickPolicy: catchup
      utc: {}
# ...
        features:
         acpi: {}
         apic: {}
         hyperv:
           frequencies: {}
           ipi: {}
           reenlightenment: {}
           relaxed: {}
           reset: {}
           runtime: {}
           spinlocks:
             spinlocks: 8191
           synic: {}
           synictimer:
             direct: {}
           tlbflush: {}
           vapic: {}
           vpindex: {}
```

VirtIO drivers

For compatibility purposes, the default Windows templates define the VM disk using the `sata` bus and the VM network using the `e1000e` bus. It is highly recommended that users install the VirtIO drivers for Windows.

When VirtIO drivers are available, using the `virtio` bus for the VM disk and the VM network interface are highly recommended for performance, example:

```
    devices:
      disks:
      - disk:
          bus: virtio
# ...
      interfaces:
      - masquerade: {}
        model: virtio
```

Compute and memory

vCPU topology

The vCPU topology determines the scheduler domain layout inside a VM, which can have performance impacts. The desired vCPU topology can be specified in the VM definition using the CPU section:

```
  cpu:
    cores: 1
    sockets: 4
    threads: 1
```

By default, if no topology is specified, `sockets` will be used for best performance since the scheduler domain used in that case includes a "sync" wakeup hint that can improve message passing performance. In general - as is the case with KVM - specifying a vCPU topology to match the host topology is typically only helpful if pinning is used. Otherwise, the default topology tends to perform well in most cases.

CPU resources

To allow for CPU overcommit by default, each VM's virt-launcher pod will define "100m" total CPU requests for the VM regardless of the vcpu definition, which is 1/10th of a host CPU from a Kubernetes resource and scheduling perspective.

For cases where overcommit is not desired, it is recommended to set the CPU resource requests near or equal to the total vCPU count to ensure the proper amount of resources are reserved from a scheduling perspective, example:

```
  cpu:
    cores: 1
    sockets: 4
    threads: 1
# ...
  resources:
    requests:
      cpu: 4
      memory: 8Gi
```

Note: when using dedicatedCpuPlacement, requests are automatically configured based on the vCPU count.

Huge pages

By default, the host kernel provides Transparent Huge Page backing for VMs and pods. For some workloads, explicitly reserving hugepages in the host and backing the guest memory with hugepages can decrease the chances of TLB (Translation Lookaside Buffer) miss and help improve performance.

See Using huge pages with virtual machines for instructions on how to configure the huge page reservation and `pageSize` backing for the VM.

Pinning

Pinning a VM can improve performance in some scenarios, by aligning the CPU and memory affinity and by providing dedicated CPUs. There are multiple levels of pinning to consider, based on workload needs and the size of the VM.

Dedicated CPU placement

Using the `dedicatedCpuPlacement` setting instructs the virt-launcher pod to automatically configure requests and limits (to qualify for a Guaranteed QoS) based on the total amount of guest CPUs and memory requests. Note that when the High-performance Workload Type is selected when creating a VM from the default templates, the dedicatedCpuPlacement setting is enabled by default.

This setting requires CPU Manager to be enabled on the node: first the virt-launcher pod will receive a cpuset of dedicated CPUs, then inside the pod libvirt will pin each guest vcpu to a host CPU in the cpuset.

Dedicated emulator thread

Further, the `isolateEmulatorThread` setting is optional to pin both the emulator thread and IOthread (when one is requested) to a single host CPU. Note that this option will add one additional CPU to the total requests and limits that the virt-launcher pod defines, and relies on `dedicatedCpuPlacement` as well.

The High-performance Workload Type also enables `isolateEmulatorThread` by default when a VM is created from the provided templates.

NUMA pinning

For workloads that are sensitive to memory affinity, the VM can be pinned to a particular NUMA node using two different methods:

- If the VM fits within a host NUMA node, the Memory Manager policy configuration on the host can be used to ensure memory affinity aligns w/ CPU affinity.
- If the VM is large enough to span multiple host NUMA nodes, the `guestMappingPassthrough` option can be used to ensure NUMA-level pinning. Note that this option requires hugepage backing of the VM memory. When specified, the host NUMA topology is mapped to the guest using libvirt numa tune definitions, based on the cpuset the virt-launcher pod is provided by CPU Manager.

Below is example syntax of all of the pinning options covered in the previous sections:

```
cpu:
  cores: 24
  sockets: 4
  threads: 1
  dedicatedCpuPlacement: true
  isolateEmulatorThread: true
  numa:
    guestMappingPassthrough : {}
```

Networking

networkInterfaceMultiqueue

For VMs with more than one vCPU, setting `networkInterfaceMultiqueue` to 'true' can improve VM network performance. This setting automatically adds queues equal to the number of vCPUs to the vhost-net device definition, allowing multiple guest CPUs to process softirq work. Note that when the High-performance Workload Type is selected when creating a VM from the default templates, `networkInterfaceMultiqueue` is enabled automatically.

Example syntax below:

```
        interfaces:
        - masquerade: {}
          model: virtio
          name: default
        networkInterfaceMultiqueue: true
```

Multus

The OpenShift SDN and OVN-Kubernetes default cluster networks provide feature-rich software-defined networking capabilities including network isolation policies, IPsec encryption, and egress firewall and router, among others.

In some cases, applications may benefit from the use of a separate data plane network option which can provide significantly better performance, especially in terms of latency. OpenShift Container Platform ships with the Mutus CNI plug-in, which allows additional networks to be configured for VMs and pods.

To configure an additional network for VMs, a Linux bridge network can be created on the host and assigned, using multus, in the VM definition. See Connecting a virtual machine to a Linux bridge network for configuration details.

OVN-Kubernetes

Note: When using the OVN cluster network, keep in mind two considerations:
Until BZ#1885605 is resolved, the host's default interface cannot be used by additional networks directly. To connect to the interface, a VLAN must first be created on a host interface before the bridge is defined, see Attach the default NIC to a bridge while using OVN Kubernetes for more information.

Bonding

As the environment allows, consider if a bond could be created on host interfaces to increase throughput capabilities for a secondary network provided to VMs, see the documentation for supported bonding modes and how to configure bonds by using a `NodeNetworkConfigurationPolicy`.

SR-IOV

For strict high-performance network requirements Single Root I/O Virtualization (SR-IOV) devices can be configured and attached to VMs, using both the SRIOV operator and Multus CNI, providing them Virtual Function(s) capable of providing near-native performance achieved by bypassing the host networking stack.

See Connecting a virtual machine to an SR-IOV network for the configuration process.

Storage

Much of the storage-related tuning is applied automatically when safe. However, there are configurable tuning options as well.

Profiles

Storage profiles are available to configure the default provisioning behavior for the configured storage class. Note that the storage profile preferences are used when the `spec.storage` API is used, but not when using `spec.pvc` API.

Note: Generally speaking, when using a storage provider that supports block volumes, block mode provides better performance than file system mode. The default `volumeMode` can be configured in a storage profile for each storage class, as in the following example using the Red Hat OpenShift Data Foundation RBD class:

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: StorageProfile
metadata:
  name: ocs-storagecluster-ceph-rbd
spec: {}
status:
  claimPropertySets:
  - accessModes:
    - ReadWriteMany
    volumeMode: Block
  - accessModes:
    - ReadWriteOnce
    volumeMode: Block
  - accessModes:
    - ReadWriteOnce
    volumeMode: Filesystem
  provisioner: openshift-storage.rbd.csi.ceph.com
  storageClass: ocs-storagecluster-ceph-rbd
```

Preallocation

For cases where write performance can be improved by preallocation or “thick” provisioning, Containerized Data Importer (CDI) provides options for preallocation methods when creating a data volume.

> Note: When importing a raw image using block `volumeMode` , by default preallocation is automatically used. If that source image is later cloned, preallocation will not be applied by default.

Disk I/O modes

Using “native” I/O mode (which uses kernel asynchronous I/O) can often provide better performance than “threads” mode (which uses user space threads), this setting can be controlled by the io option in the VM disk definition:

```
          disk:
            bus: virtio
          io: native
```

However this option can cause issues if used with a sparse disk since it can block the I/O event loop when a write happens to a not fully allocated disk and filesystem metadata needs to be updated. Because of this, OpenShift Virtualization will automatically add `io: native` when a block device or preallocated disk is used, which prevents the user from having to explicitly define this mode.

IOThreads

IOThreads are an option to improve storage performance, especially when a VM has multiple disks, by providing either a dedicated or shared thread to improve performance and scalability of block I/O requests.

By default, if a VM requests any `dedicatedIOThread` the policy is set to "shared", meaning the thread would be shared by all disk devices in the VM unless each device specifies another `dedicatedIOThread`. In cases where a VM has multiple data disks that need higher performance, the "auto" policy can be used to assign a pool of dedicated threads (up to twice the number of total vcpus) automatically to each disk device.

Note that if the High-performance Workload Type is selected when using a VM template, by default the IOthread policy is set to shared.
Example syntax below:

```
 domain:
    ioThreadsPolicy: shared
[...]
    devices:
      - disk:
          bus: virtio
        name: datadisk
        dedicatedIOThread: true
```

Note: when using IOThreads, if the VM compute resources are pinned using `dedicatedCpuPlacement`, the `isolateEmulatorThread` option can also be considered to pin the IOthread to a separate host cpu. This option will add one additional cpu to the pod definition to allow pinning of these threads and to keep them from sharing a pinned cpu with guest vcpus.

Virtio-blk multi-queue

When using very fast storage devices (for example, a device that can provide 10us read latency or less), enabling blockMultiQueue can improve I/O performance, especially for smaller block sizes. Example:

```
  devices:
     blockMultiQueue: true
     disks:
     - disk:
         bus: virtio
```

Note that `blockMultiQueue` requires a cpu request in the VM so that the number of queues can be properly configured at boot time.

1. Subject to the limits specified in Virtualization limits for Red Hat Enterprise Linux with KVM. ↩ ↩

2. Cluster sizes of around 100 nodes or less are generally recommended, for larger node counts consider multi-cluster management using Red Hat Advanced Cluster Management. Larger clusters increase upgrade timing and complexity and can cause increasing control plane stress depending on master node sizing and total object density. Multi-cluster management provides an alternative to scaling out a single control plane to very large node counts, with additional benefits around per-cluster isolation and high availability. ↩

3. The total number of VMs depends on the host hardware and resource needs ↩

Private Notes

updated 4.14 doc links

Duplicate(s) Info

Duplicates

- Supported Limits for OpenShift Virtualization 4.x

SBR Shift Virtualization **Product(s)** Red Hat OpenShift Container Platform

Category Performance tune **Component** cnv

Tags performance performance_tuning **Internal Tags** cnv **Article Type** General

---

People who viewed this article also viewed

OpenShift Container Platform Best Practices & Performance Tuning

Solution - Mar 13, 2017

Is there a tuned profile available for Spectrum Scale Erasure Code Edition Server?

Solution - May 19, 2022

LevelDB Tuning Guide for JBoss A-MQ

Solution - Nov 17, 2014

Case Links (Red Hat Internal)

03569588 - Salesforce / CaseView+ - rhn-support-mhougaar

03519375 - Salesforce / CaseView+ - rhn-support-josgutie

03505782 - Salesforce / CaseView+ - rhn-support-gveitmic

03462853 - Salesforce / CaseView+ - rhn-support-suagarwa

Comments

Add comment

Formatting Help

Send notifications to content followers

Mark comment as private

Submit