TUNING GUIDE
AMD EPYC 7003

Kubernetes Containers

Publication	57084
Revision	3.0
Issue Date	Mar, 2022

Date	Version	Changes
Mar, 2021	2.0	Initial public release
Mar, 2022	3.0	• **Modified**: NUMA Topology Manager policy to Restricted • **New feature:** SMT Awareness • **New feature:** NUMA-Aware Memory Manager • **Added:** Node Pressure Eviction • **Added:** Max Pod Threshold

Audience

This tuning guide is intended for a technical audience such as production deployment and performance engineering teams with:

- A background in configuring servers.
- Administrator-level access to both the server management Interface (BMC) and the OS.
- Familiarity with setting up Kubernetes clusters.
- Familiarity with both the BMC and OS-specific configuration, monitoring, and troubleshooting tools.

Author

Sonemaly Phrasavath

Note: All of the settings described in this Tuning Guide apply to all AMD EPYC 7003 Series Processors with or without AMD 3D V-Cache™ except where explicitly noted otherwise.

Table of Contents

This page intentionally left blank.

Chapter 1 Introduction

Containerizing an application platform and its associated dependencies abstracts the underlying infrastructure and OS differences for efficiency. Each container is bundled into one package containing an entire runtime environment, including an application with all its dependencies, libraries and other binaries, and configuration files needed to run that application. Containers running applications in a production environment need management to ensure consistent uptime. If a container goes down, then another container needs to start automatically.

Kubernetes (K8S) enables automated container deployment and management. According to What is Kubernetes?*, "K8S is a portable, extensible, open-source platform for managing containerized workloads and services. It takes care of scaling and failover for your application, provides deployment and more. It has a large, rapidly growing ecosystem. K8S services, support, and tools are widely available."

Kubernetes includes the following features:

- Service discovery and load balancing.
- Storage orchestration.
- Automated rollouts and rollbacks.
- Automatic bin packing.
- Self-healing.
- Secret and configuration management.

This tuning guide provides detailed descriptions of Kubernetes configuration settings that can optimize containerized application performance on servers powered by AMD EPYC™ 7003 Series processors.

1.1 About Tuning Kubernetes

Workloads that scale and perform well on bare metal should see a similar scaling curve in a container environment with minimal performance overhead. Some containerized workloads can even see close to 0% performance variance compared to bare metal. Large overhead generally means that application settings and/or container configuration are not optimally set. These topics are beyond the scope of this tuning guide. However, the CPU load balancing behavior of Kubernetes or other container orchestration platform scheduler may assign or load balance containerized applications differently than in a bare metal environment.

For example, a test that deployed multiple concurrent containers with a 1 CPU resource limit using Docker Swarm saw the scheduler filling the physical cores before filling the logical cores. The Linux scheduler did the same thing when the same set of containers was manually deployed. By contrast, the Kubernetes scheduler sometimes filled the logical cores before the physical cores. This unexpected behavior can be interpreted as lowering performance compared to containers with an identical workload running on a physical core. In another example, the Kubernetes scheduler varies from the Linux scheduler by sometimes deploying a pod with multi-CPU resource limits across multiple CCX/CCDs or NUMA nodes.

Please see the latest version of Overview of AMD EPYC™ 7003 Series Processors Microarchitecture for detailed information about CCXs, CCDs, and NUMA references. All three environments (Docker Swarm, Kubernetes, and Linux) implement the Completely Fair Scheduling (CFS) method but differ in how they schedule applications and their threads to physical or logical cores.

This tuning guide focuses on recommended Kubernetes environment configuration settings that can help mitigate unexpected scheduler behaviors by assigning application threads to physical and logical cores in a manner similar to a Linux environment. Additional application-specific BIOS and other settings may further boost performance. See the latest version of the appropriate application-specific tuning guide for additional information.

1.2 AMD EPYC 7003 Series Processors

AMD EPYC 7003 Series Processors are built with the leading-edge "Zen 3" core and AMD Infinity Architecture. The AMD EPYC SoC offers a consistent set of features across 8 to 64 cores. Each 3rd Gen EPYC processor consists of up to eight Core Complex Dies (CCD) and an I/O Die (IOD). Each CCD contains one CCX, meaning that each CCD contains up to 8 "Zen 3" cores. The CCDs connect to the I/O Die (IOD) to access memory, I/O, and each other via AMD Infinity Fabric™ technology. 3rd Gen AMD EPYC processors support up to 8 memory channels, 4 TB of high-speed memory per socket, and 128 lanes of PCIe® Gen 4.

3rd Gen AMD EPYC Series processors are built with the following specifications:

3rd Gen AMD EPYC 7003 Series Processors	
Process technology	7nm
Max Processor speed	4.1 GHz
Max number of cores	64
Max memory speed	3200 MT/s
Max memory capacity	4 TB per socket
Peripheral Component Interconnect	128 lanes (max) PCIeGen4

Table 1-1: Table 1 AMD EPYC™ 7003 Series Processors

Some AMD EPYC™ 7003 Series Processors introduce AMD's new 3D V-Cache die stacking technology that enables denser, more efficient chiplet integration. AMD 3D Chiplet architecture stacks L3 cache tiles vertically to provide 768 MB of L3 cache per socket up with to 96MB of L3 cache per CCD, while still providing socket compatibility with existing AMD EPYC 7003 Series Processors. Applications that take advantage of AMD 3D V-cache can see significant performance gain and lower overall Total Cost of Ownership (TCO).

See *Overview of AMD EPYC™ 7003 Series Processors Microarchitecture* (available from AMD EPYC Tuning Guides) to learn more about the AMD EPYC 7003 Series Processor microarchitecture.

1.3 Operating Systems

AMD recommends using the latest available version of the Operating Systems. Please see AMD EPYC™ 7003 Series Processors Minimum Operating System (OS) Versions for detailed OS version information.

Chapter 2

Best Practices for Container Deployment

This chapter recommends best practices for container deployment on Kubernetes

2.1 General

- **Increase max-pods for AMD EPYC Processor SKUs that can handle more density.** The default maximum number of pods that can run on a worker node is 110. AMD EPYC processors can potentially handle many more than this depending on container resource sizing. Modify the `–max-pods` kubelet flag to take advantage of the compute, memory, and IO density found in some AMD EPYC SKUs.

- **Reserve CPUs and memory resources for kubelet and system daemons and eviction thresholds.** Kubelet services run on every Kubernetes node in a cluster. Worker nodes communicate with the Controller node by sending heartbeats every few seconds to determine node availability and pass other health check data. The kubelet service therefore uses CPU resources even when no workload is running. A Kubernetes node can be scheduled to capacity, meaning that pods can consume all available capacity and leave few resources for system daemons that power the OS and Kubernetes itself. Competition between pods and the daemons could cause node resource starvation and issues unless sufficient resources are allocated to these tasks. Kubelet can use eviction thresholds to proactively terminate pods and reclaim resources. Kubelet monitors node resources such CPU, memory, and filesystem I/O on the nodes and can fail one or more pod(s) on that node when one or more resources reach a specific consumption level, thereby reclaiming resources and preventing starvation.

- **Implement pod resource limits and enable SMT awareness to reduce noisy neighbor scenarios:** Each container has unlimited access to host CPU cycle by default, which means that a "noisy neighbor" container can drain resources from another container on the same host. You can mitigate this problem by limiting the CPU, memory, and/or I/O resources available to each container in the pod definition file. Defining container resource requests and limits determines the Quality of Service (QoS) class of the pod. Kubernetes automatically classifies pods into one of three QoS categories:

  - **Guaranteed:** Pods receive this classification when the container in the pod has a CPU and memory request and limit. The requests and limits must be equal.

  - **Burstable:** Pods receive this classification when it does not meet the **Guaranteed** QoS class and has a container with either a memory or CPU request.

  - **BestEffort:** Pods receive this classification if the container has no CPU or memory request or limit. **BestEffort** pods have the lowest scheduling priority and could be evicted to make room for **Guaranteed** or **Burstable** pods.

  On a SMT enabled node, kubelet treats physical and logical cores with the same scheduling priority. SMT threads are sometimes scheduled even when physical cores are available. Resource contention typically occurs on SMT threads, which can seem like a performance regression because the container is running on a thread as opposed to a physical core. In some cases, pods could be assigned virtual and physical cores that are not siblings, which can cause different containers to share a physical core and contribute to a noisy neighbor problem. Kubernetes 1.23 has a beta feature to modify the `cpu-manager-policy-options` flag, When this flag is set to `full-pcpus-only=true`, the static policy option will always allocate full physical cores. Kubelet will only admit pods if the entire CPU request for all containers can be fulfilled by allocating full physical cores. For example, kubelet will not admit a **Guaranteed** pod with 1 CPU in an SMT environment. The result will generate a SMTAlignmentError message.

- **Implement CPU pinning for better performance:** Kubernetes uses CFS to enforce pod CPU limits. Again, a container has unlimited access to host CPU cycles by default. A single compute node in a Kubernetes cluster can run multiple pods, some of which may be running CPU-intensive workloads. Pods in this scenario might contend for the CPU resources available to that compute node. The workload can move (load balance) to different CPUs when the level of contention rises depending on whether the pod is throttled and the current CPU availability at scheduling time. Some workloads have no problem with this; however the scheduler may spend more time load balancing than processing the application if the system is heavily utilized. Some workloads may be sensitive to context switches. Both scenarios may impact workload performance, and pinning the containers may be helpful. This feature is disabled by default.

- **Implement Node level NUMA topology alignment for CPU and PCI devices:** CPU pinning alone is not enough for latency-sensitive workloads. For example, a multi-threaded, network-I/O-intensive workload may be pinned to CCXs or CCDs from different NUMA nodes. By default, Kubernetes does not guarantee that a container application will be assigned resources that are local to a NUMA node. The Topology Manager provides the Hint Provider interface for Kubernetes components to send and receive topology information. This acts as a source of truth that allows other kubelet components to make topology-aligned resource allocation choices. Topology Manager aligns the resources requested by Hint Provider so as to assign CPU and I/O to the container or pod from the same NUMA node. The Topology Manager provides two distinct settings:

  - **Scope:** Aligns resources at the pod or container level.

  - Defines how the alignment is carried out, which will be `best-effort`, `restricted`, or `single-numa-node`.

- **Use NUMA-Aware Memory Manager for platforms with NUMA nodes greater than one:** Get the best performance and latency for your workload by aligning container CPUs, peripheral devices, and memory to the same NUMA locality. Kubernetes versions prior to v1.22 included a kubelet with a NUMA topology manager that aligned CPUs and PCI devices, but not memory. The Linux kernel made best-effort attempts to allocate memory for tasks from the same NUMA node where the executing container was placed but could not guarantee that placement.

2.2 Hardware Configuration

Kubernetes can theoretically support up to 5,000 nodes. Think of a Kubernetes cluster as a "super node/machine" that is an abstraction of sets of individual nodes. The total cluster compute capacity is the sum of the CPU and memory in the individual nodes.

Every node must be able to communicate with every other node. The control plane manages this communication. The Kubernetes manager regularly iterates through all the nodes in the cluster to run a health check. More nodes mean a higher controller node.

A Kubernetes cluster includes two main components: a control plane and worker nodes. In production environments, the control plane usually runs across multiple computers, and a cluster usually runs multiple nodes, thereby providing fault-tolerance and high availability.

Note: Add-on Kubernetes components such as health monitoring are beyond the scope of this tuning guide.

2.2.1 Controller (Control Plane)

The control plane manages the worker nodes and the pods in the cluster. The control plane components make global decisions about the cluster such as scheduling and detecting and responding to cluster events. Here are a few examples of controller node sizes used by cloud providers, and you can find more information at Architecting Kubernetes clusters – Choosing a Worker Node Size*.

- **Google Compute Platform:**
    - **5 nodes (n1-standard-1 master nodes):** 6.5 GiB of memory, 1 vCPU.
    - **500 nodes (n1-standard-32 master nodes):** 120 GiB of memory, 32 vCPU.
- **AWS:**
    - **5 nodes (m3.medium master nodes):** 3.75 GiB of memory, 1 vCPU.
    - **500 nodes (c4.8xlarge master nodes):** 60 GiB of memory, 64 vCPU.

2.2.2 Worker Nodes

Nodes run containerized applications. The Kubelet, kube-proxy, and container runtime components run on every node to maintain running pods and provide the Kubernetes runtime environment.

Determine the cluster compute need, and then consider what type of worker nodes to deploy. There are pros and cons around deploying a larger number of less-powerful worker nodes instead versus fewer nodes with more CPU, memory, and I/O capacity. For example, if the total cluster compute need is 128 CPU and 8TB of memory, then you can deploy either 2 nodes with 64 CPU and 4TB of memory each or 4 nodes with 32 CPU and 2TB of memory each. The type of applications being deployed and other factors are described in Architecting Kubernetes clusters – Choosing a Worker Node Size*.

2.3 Testing Kubernetes Scheduler CPU Resource Assignment

AMD used a test environment with one worker node to observe how the Kubernetes scheduler assigned CPU resources for Guaranteed and Burstable QoS-class pods. Testing did not consider BestEffort QoS-class pods because they are the lowest-priority pod type. Table 2-1 describes the test environment.

2.3.1 Software Configuration

Table 2-1 lists the software configurations used for testing.

Name	Version	Description
BIOS OPTION	N/A	• NUMA per SOCKET=2 • SMT = ON
Ubuntu	20.04	OS distribution
Kernel	5.4.0-58-generic	Host OS
Kubernetes	1.23.3	Container orchestration platform
Docker CE	19.03.13	Container runtime environment
Containerd	1.3.7	Container runtime that abstracts system calls or OS-specific functions to run containers on Linux, Windows, or other operating systems.
Runc	1.0.0-rc10	Container runtime environment
Multi-threaded Kernel Compilation https://hub.docker.com/_/gcc	gcc:latest tag	Multi-threaded kernel compilation script that leverages the official gcc container image on Docker Hub.

Table 2-1: Software configuration

Ubuntu Container Image	16.04	Base OS image for running Sysbench
Sysbench	0.4.12	CPU-intensive single-thread application running on an Ubuntu container image used for testing Kubernetes scheduler behavior under various tuning conditions.

Table 2-1: Software configuration (Continued)

2.3.2 Test Methodology

The tuning knobs suggested in "General" on page 3 can be categorized as tuning for either node stability or performance. These tests enabled each performance-related tuning knob, such as CPU Pinning, NUMA Topology Manager, SMT Awareness, and NUMA-Aware Memory Manager one at a time. A validation run occurred after enabling each knob. Each test run deployed Sysbench and GCC pods to observe kublet behavior, with workload metrics collected to measure the effect or benefits. Htop was used to visually verify that pods were correctly assigned to CPU resources corresponding to the kubelet flag enabled.

GCC pods run a Linux kernel compilation workload utilizing more than two parallel threads. Compilation time is the metric for a GCC pod. The Sysbench workload is a single-threaded CPU intensive test. A Sysbench container was deployed using two types of Guaranteed pods: one with a 1CPU resource requirement, and another for 2CPUs.

In general:

- Guaranteed pods using a static cpu-manager policy performed better.
- Setting `Numa-topology-manager` to `restricted` policy improved Guaranteed pod performance.
- Enabling SMT Awareness resulted in consistent performance across multiple Guaranteed pods.
- Enabling NUMA-Aware Memory Manager did not make much difference, but this could be related to the type of workload used for testing.

Validating node stability knob settings is typically a function of pod density. You will see examples of how to set those values in subsequence sections of this tuning guide.

Chapter 3 Recommended Settings

Kubernetes provides a few tuning knobs to optimize containerized workload deployment. These settings affect scheduler behavior of Guaranteed QoS pods. The following sections describe how to enable those tuning knobs and other settings that assist with node stability. These settings are all associated with the kublet service and modifications are contained in one file.

3.1 Reserving CPU for Kubelet and System Daemons

`Reserved-cpus` is a kubelet flag that defines an explicit CPU set for the OS system daemons and Kubernetes system daemons. For example:

```
# vi /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

Append `-reserved-cpus` to the `ExecStart` parameter.

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-3,128-131 \
$KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS
$KUBELET_EXTRA_ARGS
```

Restart the kubelet service.

```
# systemctl restart kubelet
```

3.2 Container Pinning Settings

By default, the scheduler load balances such that the containerized application will bounce around to different CPUs. Set CPU Manager to `static` to pin the pods or containers.

Append the option `-- cpu_manager=static` to the `ExecStart` parameter

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-3,128-131 \
--cpu-manager-policy=static \
$KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS
$KUBELET_EXTRA_ARGS
```

Delete the `cpu_manager_state` file before restarting the kubelet service.

```
# rm /var/lib/kubelet/cpu_manager_state
```

Restart the kubelet service.

3.3 Resolving the Noisy Neighbor Problem

3.3.1 Pod Resource Limits

Defining a pod resource limit mitigates some noisy neighbor problem. It is good practice to set pod limits. Here are some examples of how to define QoS class for pods:

- Pods are given Guaranteed QoS class if their CPU and memory request match their limits.

```
apiVersion: batch/v1 kind: Job
metadata:
  name: sysbench1
  namespace: sysbench
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: sysbench1 image: sb-test:latest
        imagePullPolicy: IfNotPresent resources:
        limits: cpu: "2"
        memory: "500M" requests:
        cpu: "2" memory: "500M"
  nodeSelector:
  name: controller
```

- Pods are given Burstable QoS class if it has a CPU or memory request.

```
apiVersion: batch/v1 kind: Job
metadata:
  name: sysbench1
  namespace: sysbench
spec:
  template: spec:
  restartPolicy: Never
    containers:
    - name: sysbench1 image: sb-test:latest
    imagePullPolicy: IfNotPresent resources:
    limits: cpu: "2"
    requests: cpu: "1"
nodeSelector:
  name: controller
```

3.3.2 SMT Alignment

Another factor that could contribute to a noisy neighbor scenario occurs when multiple pods share the same physical CPUs. The static cpu-manager policy for kubelet includes an option that forces Guaranteed pods to use only full physical cores.

Append `--cpu-manager-policy-options="full-pcpus-only=true"` to the `ExecStart` parameter.

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-3,128-131 \
--cpu-manager-policy=static \
--cpu-manager-policy-options="full-pcpus-only=true" \
$KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS
```

Restart the kubelet service. If the service doesn't start because it failed to initialize the CPU manager policy, then delete `/var/lib/kubelet/cpu_manager_state`, and then restart the service.

3.4 NUMA Alignment Settings

CPU pinned containers can be assigned resources from different NUMA nodes. Enabling Topology Manager and setting it to **Restricted** forces the kubelet to align CPU, memory, or I/O resources to the same NUMA locality without limiting the pod/container to just `single_numa_node`. The **Restricted** policy limits the preferred kubelet alignment to the minimum possible NUMA nodes for a given request size on a given machine. For example, consider a system with the following NUMA configuration:

```
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 64 65 66 67 68 69 70 71 72 73 74 75 76
77 78 79
node 0 size: 128642 MB
node 0 free: 125955 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95
node 1 size: 128974 MB
node 1 free: 127234 MB
node 2 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 96 97 98 99 100 101 102 103 104
105 106 107 108 109 110 111
node 2 size: 129014 MB
node 2 free: 128342 MB
node 3 cpus: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127
```

- A Guaranteed pod with a CPU request size less than or equal to 32 will be restricted to a single NUMA node.
- A Guaranteed pod with a CPU request size equal to or greater than 32 but less than 64 will be restricted to 2 NUMA nodes.

In this example, a `single_numa_node` policy will not admit the second pod. See "NUMA Aware Memory Manager" on page 10.

Append `--topology-manager-policy=single-numa-node` to the `ExecStart` parameter.

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-3,128-131 \
--cpu-manager-policy=static \
--cpu-manager-policy-options="full-pcpus-only=true" \
--topology-manager-policy=restricted \
$KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS
$KUBELET_EXTRA_ARGS
```

Restart the kubelet service.

```
# systemctl kubelet restart
```

3.5 NUMA Aware Memory Manager

Memory Manager provides guaranteed memory and hugepages allocation for Guaranteed QoS class pods. Other policy managers should be set prior to the Memory Manager. This policy has an accompanied flag called `–reserved-memory`.

If the policy is set to **static** and other node allocatable mechanisms such as `–eviction-threshold` are utilized, then `--reserved-memory` is mandatory.

"Node Stability Settings" on page 11 shows the `–eviction-threshold` for memory is set to 1Gi. Therefore, set the following `memory-manager-policy` and **reserved-memory** values:

Append `--memory-manager-policy` and `--reserved-memory` to the `ExecStart` parameter.

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-3,128-131 \
--cpu-manager-policy=static \
--cpu-manager-policy-options="full-pcpus-only=true" \
--topology-manager-policy=single-numa-node \
--memory-manager-policy=Static \
--reserved-memory="0:memory=1Gi" \
$KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS
$KUBELET_EXTRA_ARGS
```

3.6 Node Stability Settings

Eviction thresholds are the minimum amount of resources that should be available on the node. Eviction signals are the current state of a particular resource at a specific time.

Kubelet uses eviction signals to make eviction decisions by comparing the signals to the thresholds. Please see `K8S node-presssure-eviction` for a list of evictions signals used by kubelet,

Append `--eviction-hard=memory.available<1Gi` to the `ExecStart` parameter.

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-3,128-131 \
--cpu-manager-policy=static \
--cpu-manager-policy-options="full-pcpus-only=true" \
--topology-manager-policy=single-numa-node \
--memory-manager-policy=Static \
--reserved-memory="0:memory=1Gi" \
--eviction-hard=memory.available<1Gi \
$KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS $KUBELET_EXTRA_ARGS
```

This page intentionally left blank.

Chapter 4 Resources

- What is Kubernetes?*
- Architecting Kubernetes Clusters – Choosing a Worker Node Size*
- Reserve Compute Resources for System Daemons*

This page intentionally left blank.

Chapter 6 Glossary

- **K8S:** Shorthand for Kubernetes.
- **Pod:** Smallest unit of work known to Kubernetes. Pod can contain one of more containers. Node – Worker machine within the K8S cluster.
- **QoS:** Quality of Service.
- **CFS:** Completely Fair Scheduler.
- **SMT:** Simultaneous Multiple Threading.

This page intentionally left blank.