AMD EPYC

TUNING GUIDE
AMD EPYC 9004

Kubernetes® Containers

Publication	58008
Revision	1.3
Issue Date	June, 2023

Date	Version	Changes
July, 2022	0.1	Initial NDA partner release
Sep, 2022	0.2	Updated BIOS information
Nov, 2022	1.0	Initial public release
Dec, 2022	1.1	Minor errata corrections
Mar, 2023‘	1.2	Added 97xx OPN and AMD 3D V-Cache™ technology information
Jun, 2023	1.3	Second public release

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 9004 Series Processors of all core counts with or without AMD 3D V-Cache™ except where explicitly noted otherwise.

Table of Contents

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™ 9004 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 "AMD EPYC 9004 Series Processors features by model" on page 3 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.

Chapter 2

AMD EPYC™ 9004 Series Processors

AMD EPYC™ 9004 Series Processors represent the fourth generation of AMD EPYC server-class processors. This generation of AMD EPYC processors feature AMD's latest "Zen 4" based compute cores, next-generation Infinity Fabric, next-generation memory & I/O technology, and use the new SP5 socket/packaging.

2.1 General Specifications

AMD EPYC 9004 Series Processors offer a variety of configurations with varying numbers of cores, Thermal Design Points (TDPs), frequencies, cache sizes, etc. that complement AMD's existing server portfolio with further improvements to performance, power efficiency, and value. Table 1-1 lists the features common to all AMD EPYC 9004 Series Processors.

Common Features of all AMD EPYC 9004 Series Processors	
Compute cores	Zen4-based
Core process technology	5nm
Maximum cores per Core Complex (CCX)	8
Max memory per socket	6 TB
Max # of memory channels	12 DDR5
Max memory speed	4800 MT/s DDR5
Max lanes Compute eXpress Links	64 lanes CXL 1.1+
Max lanes Peripheral Component Interconnect	128 lanes PCIe® Gen 5

Table 2-1: Common features of all AMD EPYC 9004 Series Processors

2.2 Model-Specific Features

Different models of 4th Gen AMD EPYC processors have different feature sets, as shown in Table 1-2.

AMD EPYC 9004 Series Processor (Family 19h) Features by Model		
Codename	"Genoa"*	"Bergamo"*
Model #	91xx-96xx	97xx
Max number of Core Complex Dies (CCDs)	12	8
Number of Core Complexes (CCXs) per CCD	1	2
Max number of cores (threads)	96 (192)	128 (256)
Max L3 cache size (per CCX)	1,152 MB (96 MB)♦	256 MB (16 MB)
Max Processor Frequency	4.4 GHz♦♦	3.15 GHz
Includes ♦AMD 3D V-Cache (9xx4X) and ♦♦high-frequency (9xx4F) models.		
*GD-122: The information contained herein is for informational purposes only and is subject to change without notice. Timelines, roadmaps, and/or product release dates shown herein and plans only and subject to change. "Genoa" and "Bergamo" are codenames for AMD architectures and are not product names.		

Table 2-2: AMD EPYC 9004 Series Processors features by model

2.3 Operating Systems

AMD recommends using the latest available targeted OS version and updates. Please see AMD EPYC™ Processors Minimum Operating System (OS) Versions for detailed OS version information.

2.4 Processor Layout

AMD EPYC 9004 Series Processors incorporate compute cores, memory controllers, I/O controllers, RAS (Reliability, Availability, and Serviceability), and security features into an integrated System on a Chip (SoC). The AMD EPYC 9004 Series Processor retains the proven Multi-Chip Module (MCM) Chiplet architecture of prior successful AMD EPYC processors while making further improvements to the SoC components.

The SoC includes the Core Complex Dies (CCDs), which contain Core Complexes (CCXs), which contain the "Zen 4"-based cores. The CCDs surround the central high-speed I/O Die (and interconnect via the Infinity Fabric). The following sections describe each of these components.

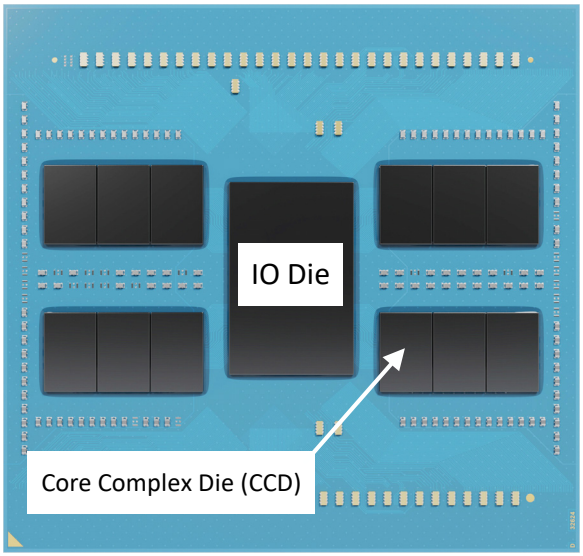

Figure 2-1: AMD EPYC 9004 configuration with 12 Core Complex Dies (CCD) surrounding a central I/O Die (IOD)

2.5 "Zen 4" Core

AMD EPYC 9004 Series Processors are based on the new "Zen 4" compute core. The "Zen 4" core is manufactured using a 5nm process and is designed to provide an Instructions per Cycle (IPC) uplift and frequency improvements over prior generation "Zen" cores. Each core has a larger L2 cache and improved cache effectiveness over the prior generation. Each "Zen 4" core includes:

- Up to 32 KB of 8-way L1 I-cache and 32 KB of 8-way of L1 D-cache
- Up to a 1 MB private unified (Instruction/Data) L2 cache.

Each core supports Simultaneous Multithreading (SMT), which allows 2 separate hardware threads to run independently, sharing the corresponding core's L2 cache.

2.6 Core Complex (CCX)

Figure 2-2 shows a Core Complex (CCX) where up to eight "Zen 4"-based cores share a L3 or Last Level Cache (LLC). Enabling Simultaneous Multithreading (SMT) allows a single CCX to support up to 16 concurrent hardware threads.

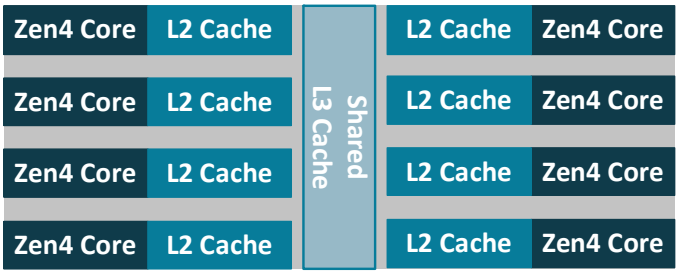

Figure 2-2: Top view of 8 compute cores sharing an L3 cache (91xx-96xx models)

2.7 Core Complex Dies (CCDs)

The Core Complex Die (CCD) in an AMD EPYC 9xx4 Series Processor may contain either one or two CCXs, depending on the processor (91xx-96xx "Genoa" vs. 97xx "Bergamo"), as shown in Figure 2-5.

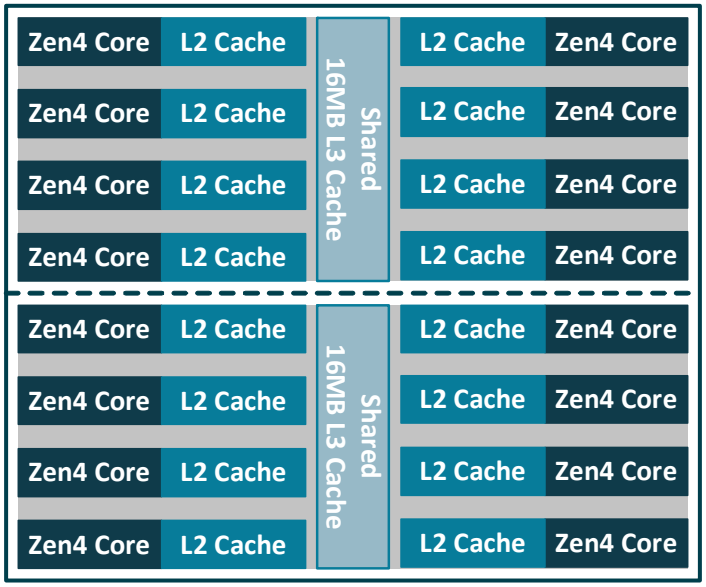

Figure 2-3: 2 CCXs in a single 4th Gen AMD EPYC 97xx CCD

Each of the Core Complex Dies (CCDs) in a 97xx model AMD EPYC 9004 Series Processor contains two CCXs (Figure 2-5):

AMD EPYC 9004 Series Processor	91xx-96xx	97xx
# of CCXs within a CCD	1	2

Table 2-3: CCXs per CCD by AMD EPYC model

You can disable cores in BIOS using one or both of the following approaches:

- Reduce the cores per L3 from 8 down to 7,6,5,4,3,2, or 1 while keeping the number of CCDs constant. This approach increases the effective cache per core ratio but reduces the number of cores sharing the cache.
- Reduce the number of active CCDs while keeping the cores per CCD constant. This approach maintains the advantages of cache sharing between the cores while maintaining the same cache per core ratio.

2.8 AMD 3D V-Cache™ Technology

AMD EPYC 9xx4X Series Processors include AMD 3D V-Cache™ die stacking technology that enables 97xx to achieve more efficient chiplet integration. AMD 3D Chiplet architecture stacks L3 cache tiles vertically to provide up to 96MB of L3 cache per die (and up to 1 GB L3 Cache per socket) while still providing socket compatibility with all AMD EPYC™ 9004 Series Processor models.

AMD EPYC 9004 Series Processors with AMD 3D V-Cache technology employ industry-leading logic stacking based on copper-to-copper hybrid bonding "bumpless" chip-on-wafer process to enable over 200X the interconnect densities of current 2D technologies (and over 15X the interconnect densities of other 3D technologies using solder bumps), which translates to lower latency, higher bandwidth, and greater power and thermal efficiencies.

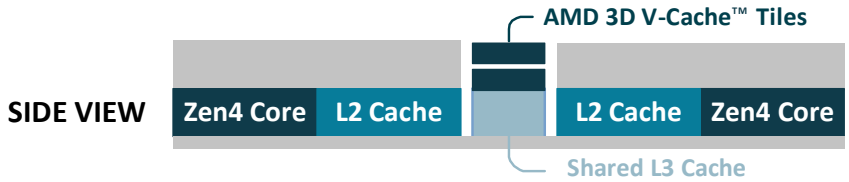

Figure 2-4: Side view of vertically-stacked central L3 SRAM tiles

AMD EPYC 9004 Series Processors	9xx4	9004X (with 3D V-Cache)
Max Shared L3 Cache per CCD	32 MB	96 MB

Table 2-4: L3 cache by processor model

Different OPNs also may have different numbers of cores within the CCX. However, for any given part, all CCXs will always contain the same number of cores.

2.9 I/O Die (Infinity Fabric™)

The CCDs connect to memory, I/O, and each other through an updated I/O Die (IOD). This central AMD Infinity Fabric™ provides the data path and control support to interconnect CCXs, memory, and I/O. Each CCD connects to the IOD via a dedicated high-speed Global Memory Interconnect (GMI) link. The IOD helps maintain cache coherency and additionally provides the interface to extend the data fabric to a potential second processor via its xGMI, or G-links. AMD EPYC 9004 Series Processors support up to 4 xGMI (or G-links) with speeds up to 32Gbps. The IOD exposes DDR5 memory channels, PCIe® Gen5, CXL 1.1+, and Infinity Fabric links.

All dies (chiplets) interconnect with each other via AMD Infinity Fabric technology. Figure 2-6 (which corresponds to Figure 2-2, above) shows the layout of a 96-core AMD EPYC 9654 processor. The AMD EPYC 9654 has 12 CCDs, with each CCD connecting to the IOD via its own GMI connection.

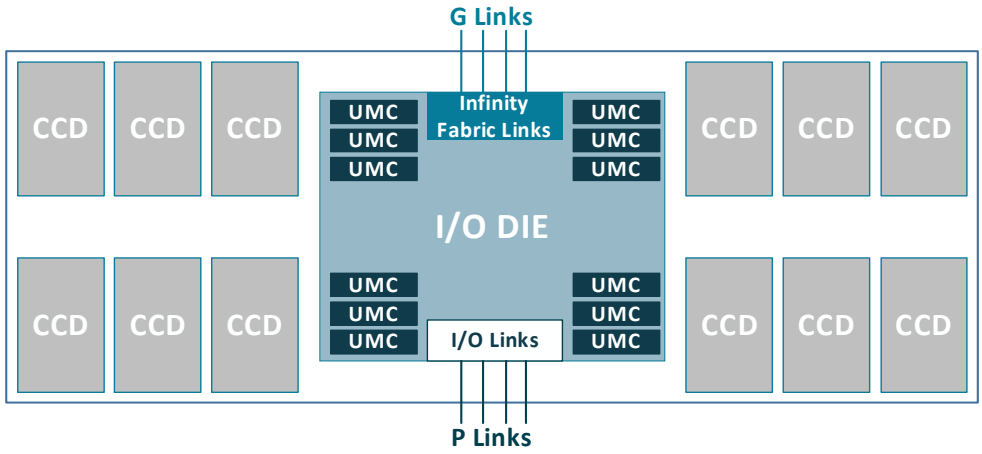

Figure 2-5: AMD EPYC 9654 processor internals interconnect via AMD Infinity Fabric (12 CCD processor shown)

AMD also provides "wide" OPNs (e.g. AMD EPYC 9334) where each CCD connects to two GMI3 interfaces, thereby allowing double the Core-to-I/O die bandwidth.

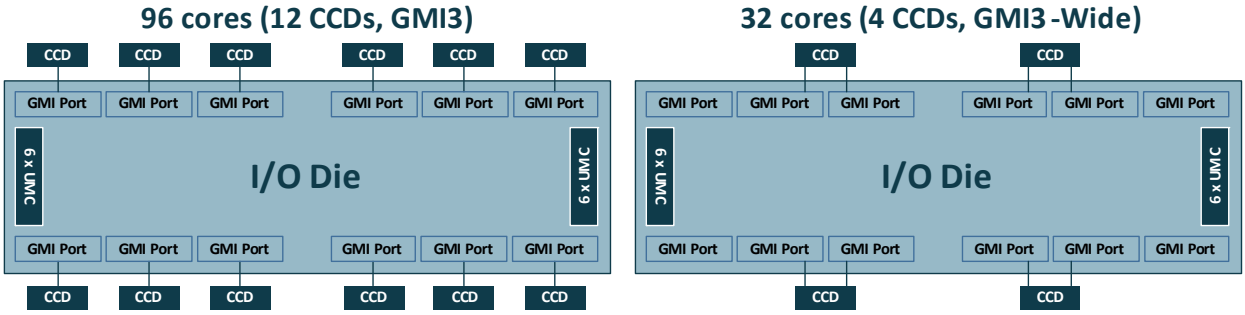

Figure 2-6: Standard vs. Wide GMI links

The IOD provides twelve Unified Memory Controllers (UMCs) that support DDR5 memory. The IOD also presents 4 'P-links' that the system OEM/designer can configure to support various I/O interfaces, such as PCIe Gen5, and/or CXL 1.1+.

2.10 Memory and I/O

Each UMC can support up to 2 DIMMs per channel (DPC) for a maximum of 24 DIMMs per socket. OEM server configurations may allow either 1 DIMM per channel or 2 DIMMs per channel. 4th Gen AMD EPYC processors can support up to 6TB of DDR5 memory per socket. Having additional and faster memory channels compared to previous generations of AMD EPYC processors provides additional memory bandwidth to feed high-core-count processors. Memory interleaving on 2, 4, 6, 8, 10, and 12 channels helps optimize for a variety of workloads and memory configurations.

Each processor may have a set of 4 P-links and 4 G-links. An OEM motherboard design can use a G-link to either connect to a second 4th Gen AMD EPYC processor or to provide additional PCIe Gen5 lanes. 4th Gen AMD EPYC processors support up to eight sets of x16-bit I/O lanes, that is, 128 lanes of high-speed PCIe Gen5 in single-socket platforms and up to 160 lanes in dual-socket platforms. Further, OEMs may either configure 32 of these 128 lanes as SATA lanes and/or configure 64 lanes as CXL 1.1+. In summary, these links can support:

- Up to 4 G-links of AMD Infinity Fabric connectivity for 2P designs.
- Up to 8 x16 bit or 128 lanes of PCIe Gen 5 connectivity to peripherals in 1P designs (and up to 160 lanes in 2-socket designs).
- Up to 64 lanes (4 P-links) that can be dedicated to Compute Express Link (CXL) 1.1+ connectivity to extended memory.
- Up to 32 I/O lanes that can be configured as SATA disk controllers.

2.11 Visualizing AMD EPYC 9004 Series Processors (Family 19h)

This section depicts AMD EPYC 9004 Series Processors that have been set up with four nodes per socket (NPS=4). Please see "NUMA Topology" on page 10 for more information about nodes.

2.11.1 Models 91xx-96xx ("Genoa")

4th Gen AMD EPYC 9004 processors with model numbers 91xx-96xx have up to 12 CCDs that each contain a single CCX, as shown below.

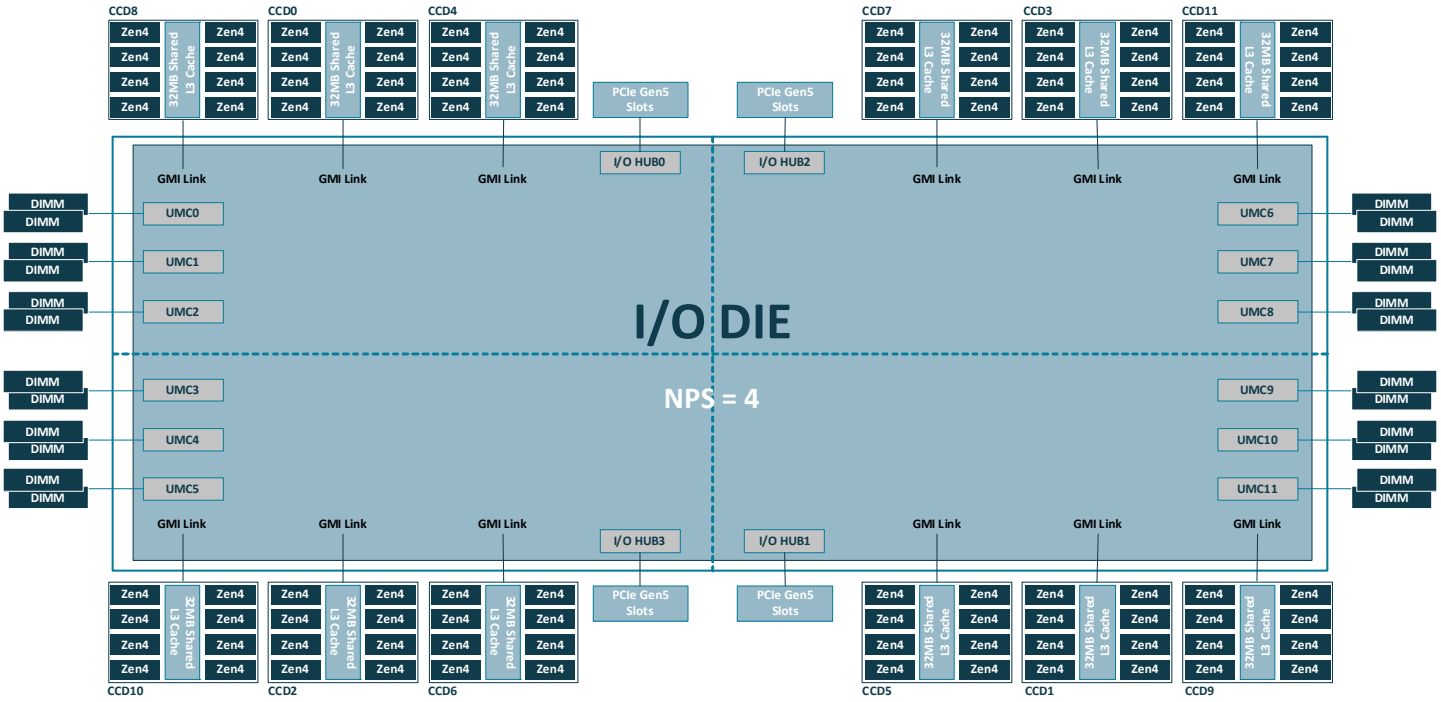

Figure 2-7: The AMD EPYC 9004 SoC consists of up to 12 CCDs and a central IOD for 91xx-96xx models, including "X" OPNs

2.11.2 Models 97xx (“Bergamo”)

97xx 4th Gen AMD EPYC 9004 Series Processors with model numbers 97xx have up to 8 CCDs that each contain two CCXs, as shown below.

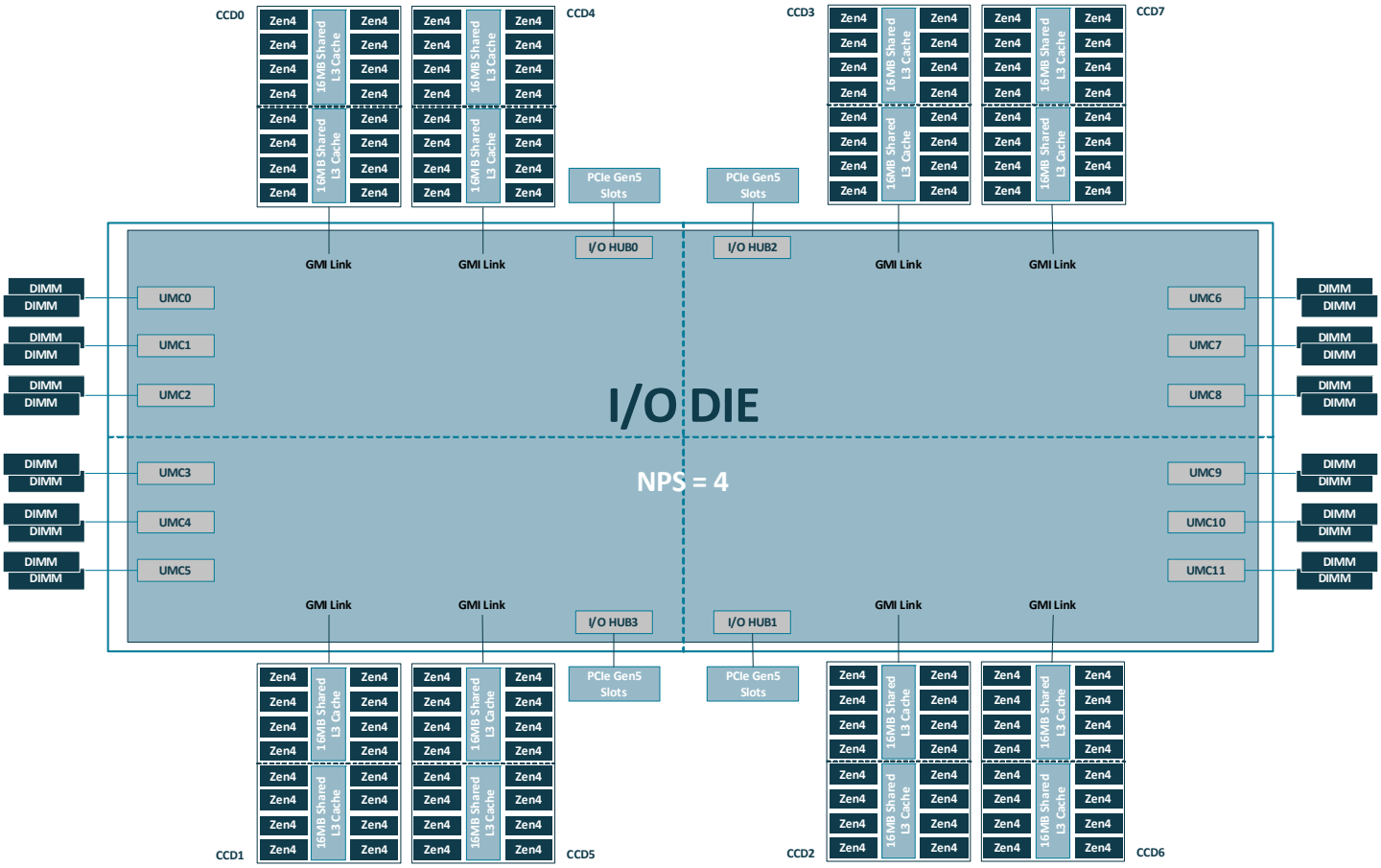

Figure 2-8: The AMD EPYC 9004 System on Chip (SoC) consists of up to 8 CCDs and a central IOD for 97xx models

2.12 NUMA Topology

AMD EPYC 9004 Series Processors use a Non-Uniform Memory Access (NUMA) architecture where different latencies may exist depending on the proximity of a processor core to memory and I/O controllers. Using resources within the same NUMA node provides uniform good performance, while using resources in differing nodes increases latencies.

2.12.1 NUMA Settings

A user can adjust the system **NUMA Nodes Per Socket** (NPS) BIOS setting to optimize this NUMA topology for their specific operating environment and workload. For example, setting NPS=4 as shown in “Memory and I/O” on page 8 divides the processor into quadrants, where each quadrant has 3 CCDs, 3 UMCs, and 1 I/O Hub. The closest processor-memory I/O distance is between the cores, memory, and I/O peripherals within the same quadrant. The furthest distance is between a core and memory controller or IO hub in cross- diagonal quadrants (or the other processor in a 2P configuration). The locality of cores, memory, and IO hub/devices in a NUMA-based system is an important factor when tuning for performance.

The NPS setting also controls the interleave pattern of the memory channels within the NUMA Node. Each memory channel within a given NUMA node is interleaved. The number of channels interleaved decreases as the NPS setting gets more granular. For example:

- A setting of NPS=4 partitions the processor into four NUMA nodes per socket with each logical quadrant configured as its own NUMA domain. Memory is interleaved across the memory channels associated with each quadrant. PCIe devices will be local to one of the four processor NUMA domains, depending on the IOD quadrant that has the corresponding PCIe root complex for that device.

- A setting of NPS=2 configures each processor into two NUMA domains that groups half of the cores and half of the memory channels into one NUMA domain, and the remaining cores and memory channels into a second NUMA domain. Memory is interleaved across the six memory channels in each NUMA domain. PCIe devices will be local to one of the two NUMA nodes depending on the half that has the PCIe root complex for that device.

- A setting of NPS=1 indicates a single NUMA node per socket. This setting configures all memory channels on the processor into a single NUMA node. All processor cores, all attached memory, and all PCIe devices connected to the SoC are in that one NUMA node. Memory is interleaved across all memory channels on the processor into a single address space.

- A setting of NPS=0 indicates a single NUMA domain of the entire system (across both sockets in a two-socket configuration). This setting configures all memory channels on the system into a single NUMA node. Memory is interleaved across all memory channels on the system into a single address space. All processor cores across all sockets, all attached memory, and all PCIe devices connected to either processor are in that single NUMA domain.

You may also be able to further improve the performance of certain environments by using the **LLC (L3 Cache) as NUMA** BIOS setting to associate workloads to compute cores that all share a single LLC. Enabling this setting equates each shared L3 or CCX to a separate NUMA node, as a unique L3 cache per CCD. A single AMD EPYC 9004 Series Processor with 12 CCDs can have up to 12 NUMA nodes when this setting is enabled.

Thus, a single EPYC 9004 Series Processor may support a variety of NUMA configurations ranging from one to twelve NUMA nodes per socket.

Note: If software needs to understand NUMA topology or core enumeration, it is imperative to use documented Operating System (OS) APIs, well-defined interfaces, and commands. Do not rely on past assumptions about settings such as APICID or CCX ordering.

2.13 Dual-Socket Configurations

AMD EPYC 9004 Series Processors support single- or dual-socket system configurations. Processors with a 'P' suffix in their name are optimized for single-socket configurations (see the "Processor Identification" chapter) only. Dual-socket configurations require both processors to be identical. You cannot use two different processor Ordering Part Numbers (OPNs) in a single dual-socket system.

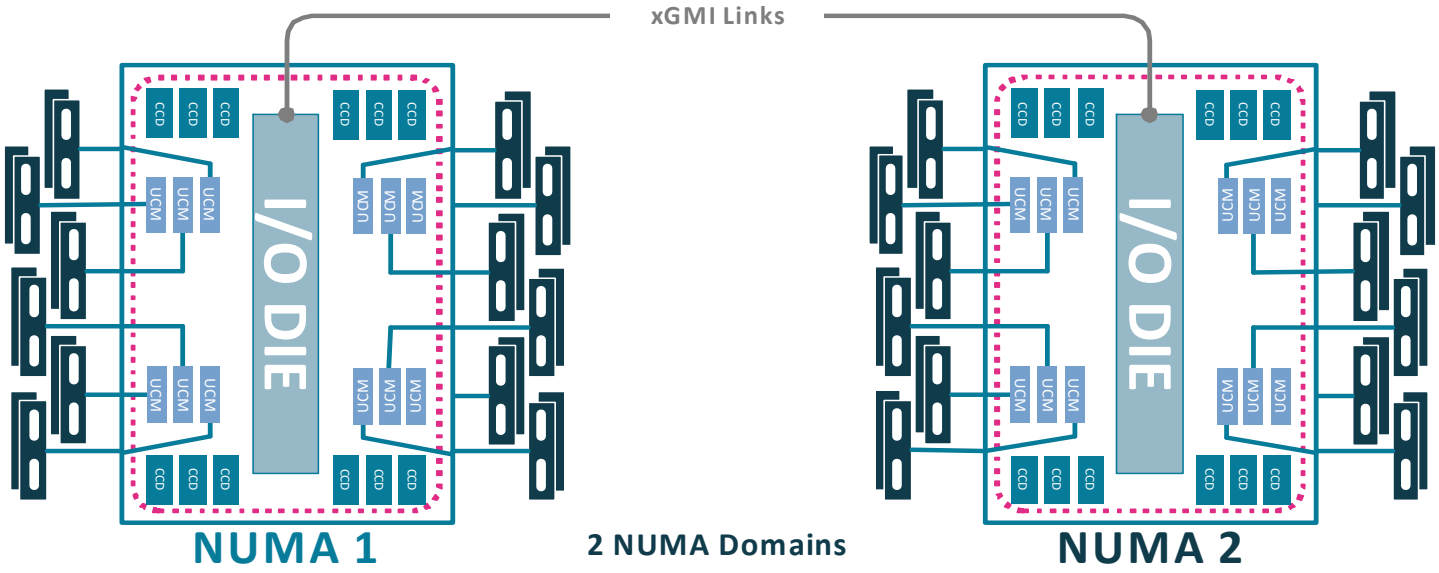

Figure 2-9: Two EPYC 9004 Processors connect through 4 xGMI links (NPS1)

In dual-socket systems, two identical EPYC 9004 series SoCs are connected via their corresponding External Global Memory Interconnect [xGMI] links. This creates a high bandwidth, low latency interconnect between the two processors. System manufacturers can elect to use either 3 or 4 of these Infinity Fabric links depending upon I/O and bandwidth system design objectives.

The Infinity Fabric links utilize the same physical connections as the PCIe lanes on the system. Each link uses up to 16 PCIe lanes. A typical dual socket system will reconfigure 64 PCIe lanes (4 links) from each socket for Infinity Fabric connections. This leaves each socket with 64 remaining PCIe lanes, meaning that the system has a total of 128 PCIe lanes. In some cases, a system designer may want to expose more PCIe lanes for the system by reducing the number of Infinity Fabric G-Links from 4 to 3. In these cases, the designer may allocate up to 160 lanes for PCIe (80 per socket) by utilizing only 48 lanes per socket for Infinity Fabric links instead of 64.

A dual-socket system has a total of 24 memory channels, or 12 per socket. Different OPNs can be configured to support a variety of NUMA domains.

Chapter

3

BIOS Defaults Summary

This chapter provides high-level lists of the default AMD EPYC 9004 BIOS settings and their default values. Please see Chapter 4 of the *BIOS & Workload Tuning Guide for AMD EPYC™ 9004 Series Processors* (available from AMD EPYC Tuning Guides) for detailed descriptions. Later chapters in this Tuning Guide discuss the BIOS options as they relate to a specific workload or set of workloads.

Note: The default setting names and values described in this chapter are the AMD default names and values that serve as recommendations for OEMs. End users must confirm their OEM BIOS setting availability and options.

AMD strongly recommends that customers download and install the latest BIOS update for your AMD EPYC 9004 Series Processor-based server from your platform vendor. BIOS updates often help customers by providing new and updated features, bug fixes, enhancements, security features, and other improvements. These improvements can help your system software stability and dependency modules (such as hardware, firmware, drivers, and software) by giving you a more robust environment to run your applications.

3.1 Processor Core Settings

Name	Default	Description
SMT Control	Auto	• **Enabled/Auto:** Two hardware threads per core. • **Disabled:** Single hardware thread per core.
L1 Stream HW Prefetcher	Auto	• **Enabled/Auto:** Enables the prefetcher. • **Disabled:** Disables the prefetcher.
L1 Stride Prefetcher	Auto	• **Enabled/Auto:** Enables the prefetcher. • **Disabled:** Disables the prefetcher.
L1 Region Prefetcher	Auto	• **Enabled/Auto:** Enables the prefetcher. • **Disabled:** Disables the prefetcher.
L1 Burst Prefetch Mode	Auto	• **Enabled/Auto:** Enables the prefetcher. • **Disabled:** Disables the prefetcher.
L2 Stream HW Prefetcher	Auto	• **Enabled/Auto:** Enables the prefetcher. • **Disabled:** Disables the prefetcher.
L2 Up/Down Prefetcher	Auto	• **Enabled/Auto:** Enables the prefetcher. • **Disabled:** Disables the prefetcher.
Core Performance Boost	Auto	• **Enabled/Auto:** Enables Core Performance Boost. • **Disabled:** Disables Core Performance Boost.
BoostFmaxEn	Auto	• **Auto:** Use the default Fmax • **Manual:** User can set the boost Fmax
BoostFmax	Auto	Specify the boost Fmax frequency limit to apply to all cores (MHz in decimal)
Global C-State Control	Auto	• **Enabled/Auto:** Controls IO based C-state generation and DF C-states, including core processor C-States • **Disabled:** AMD strongly recommends not disabling this option because this also disables core processor C-States.

Table 3-1: Processor core BIOS settings

X3D	Auto	Enables or disables AMD 3D V-Cache™ technology on Cache Optimized (9004X) processors. • **Auto:** Enabled on an AMD EPYC 9004 Series processor with AMD 3D V-Cache™ technology, enabling this option enables the AMD 3D V-Cache module in the CCD to increase the total size of the L3 cache memory to 96MB • **Disabled:** Disabling this option reduces the L3 cache in the CCD to 32MB. *Note: This option only applies to AMD EPYC 9004 Series Processors with AMD 3D V-Cache technology.* *Note: AMD engineers performed extensive internal testing and validation for various applications using the X3D BIOS option found in AMD EPYC 9xx4X processors with AMD 3D V-Cache technology. This testing and validation cannot cover all applications or use cases. Testing has shown AMD 3D V-Cache to be beneficial for most workloads, however AMD recommends that you test and evaluate the benefits of enabling or disabling the X3D BIOS option for your application workloads in your environment and proceeding based on those results.*

Table 3-1: Processor core BIOS settings

3.2 Power Efficiency Settings

Name	Default	Description
Power Profile Selection	Auto	• **Auto/0:** High-performance mode • **1:** Efficiency mode • **2:** Maximum I/O performance mode
Determinism Control	Auto	• **Auto:** Use default performance determinism settings. • **Manual:** Specify custom performance determinism settings.
Determinism Enable	Auto	• **Auto:** Performance. • **1:** Power.
TDP Control	Auto	• **Auto:** Use platform- and OPN-default TDP. • **Manual:** Set custom configurable TDP.
TDP	OPN Max	This option appears once the user sets the **TDP Control** to **Manual**. • **Values 85-400:** Set configurable TDP, in watts.
PPT Control	Auto	Enables or disables the **PPT** control. • **Auto:** Automatically set PPL in watts. • **Manual:** Specify a custom PPL.
PPT	OPN Max	This option appears once the user sets the **PPT Control** to **Manual**. • **Values 85-400:** Set configurable PPT, in watts.
CPPC	Auto	• **Enabled/Auto:** Allows the OS to make performance/power optimization requests using ACPI CPPC. • **Disabled:** Prevents the OS from making performance/power optimization requests using ACPI CPPC.

Table 3-2: Power efficiency BIOS settings

3.3 NUMA and Memory Settings

Name	Default	Description
LLC as NUMA Domain (ACPI SRAT L3 Cache as NUMA Domain)	Disabled	• **Disabled (recommended):** Both NUMA nodes (`cpubind`) and memory interleaving (`membind`) are determined by the NPS setting. • **Enabled:** Overrides the NPS setting for # of NUMA nodes by mapping each LLC as a NUMA node. This does not impact the memory interleaving
Nodes Per Socket (NPS)	1	**Memory Interleaving:** The **NPS** setting always determines the memory interleaving regardless of whether **LLC as NUMA** is **Enabled** or **Disabled**. # of NUMA nodes (if **LLC as NUMA Domain** is **Disabled**): • **NPS1/Auto:** One NUMA node per socket (Most cloud providers use this as it provides consistent average memory latency to all the accesses within a socket). • **NPS2:** Two NUMA nodes per socket. • **NPS4:** Four NUMA nodes per socket • **NPS0 (not recommended):** Only applicable for dual-socket systems. A single NUMA node is created for the whole two-socket platform. AMD recommends either NPS1 or NPS4 depending on your use case. **Windows systems:** Make sure that the number of logical processors per NUMA node is <=64. You can do this by using NPS2 or NPS4 instead of the default NPS1.
Memory Target Speed	Auto	• **Auto:** Determine the maximum memory speed based on SPD information from populated DIMMs and platform memory speed support. Alternatively, you can select: • **Values 3200–5600 MT/s:** Run the DRAM memory target clock speed at the specified speed. The DRAM memory target is the DDR rate. Your OEM system default value may vary.
Memory Interleaving	Auto	• **Auto/Enable:** Enables memory interleaving. • **Disable:** Allows for disabling memory interleaving. The N**UMA Nodes per Socket** setting will be honored regardless of this setting. AMD strongly recommends not disabling this setting because most production deployments benefit from memory interleaving.

Table 3-3: NUMA and memory BIOS settings

3.4 Infinity Fabric Settings

Name	Default	Description
3-4 xGMI Link Max Speed	Auto	• 12 Gbps • 16 Gbps • 17 Gbps • 18 Gbps • 20 Gbps • 22 Gbps • 23 Gbps • 24 Gbps • 25 Gbps/Auto • 26 Gbps • 27 Gbps • 28 Gbps • 30 Gbps • 32 Gbps Your OEM system default value may vary.
xGMI Link Width Control	Auto	• **Auto:** Use the default xGMI link width controller settings. • **Manual:** Specify a custom xGMI link width controller setting.
xGMI Force Link Width Control	Auto	• **Unforce:** Do not force the xGMI to a fixed width. • **Force:** Use the xGMI link to the user-specified width.
xGMI Force Link Width	Auto	• **0:** Force xGMI link width to x4. • **1:** Force xGMI link width to x8. • **2:** Force xGMI link width to x16.
xGMI Max Link Width Control	Auto	• **Auto:** Use the default xGMI link width controller settings. • **Manual:** Specify a custom xGMI link with controller setting.
xGMI Max Link Width	Auto	• **0:** Set max xGMI link width to x8. • **1:** Set max xGMI link width to x16.
APBDIS	Auto	• **0/Auto:** Dynamically switch the Infinity Fabric P-state based on link usage. • **1:** Enabled fixed Infinity Fabric P-state control.
DfPstate Range Support	Auto	• **Auto:** If this feature is enabled, the range value setting should follow the rule that MaxDfPstate<=MinDfPstate. Otherwise, it will not work. • **Enable:** Add the values MaxDfPstate & MinDfPstate. • **Disable:** No MaxDfPstate & MinDfPstate option.

Table 3-4: Infinity Fabric BIOS settings

DF C-States	Auto	Controls DF C-states. • **Disabled:** Prevents the AMD Infinity Fabric from entering a low-power state. • **Enabled/Auto:** Allows the AMD Infinity Fabric to enter a low-power state.

Table 3-4: Infinity Fabric BIOS settings

3.5 PCIe, I/O, Security, and Virtualization Settings

Name	Default	Description
Local APIC Mode	Auto(0x02)	• **xAPIC:** Use xAPIC, supports up to 255 cores. • **x2APIC:** Supports more than 255 cores. • **Auto:** The system will choose the mode that best fits the number of active cores in the system. • **Compatibility:** Threads below 255 run in xAPIC with xAPIC ACPI structures, and threads 255 and above run in x2 mode with x2 ACPI structures. • **XApicMode (0x01):** Forces legacy xAPIC mode. • **X2ApicMode (0x02):** Forces x2APIC mode independent of thread count.
PCIe Speed PMM Control	Auto	• **0:** Dynamic link speed determined by power management functionality. • **1:** Static Target Link Speed (Gen4); sets the maximum idle link speed to 16 GT/s. • **Auto/2:** Static Target Link Speed (Gen5); sets the maximum idle link speed to 32 GT/s, thereby disabling the feature).
PCIe ARI Support (SRIOV)	Auto	• **Enabled/Auto:** Enables Alternative Routing ID interpretation. • **Disabled:** Disables Alternative Routing ID interpretation.
PCIe Ten Bit Tag Support	Auto	• **Enabled/Auto:** Enables PCIe 10-bit tags for supported devices. • **Disabled:** Disables PCIe 10-bit tags for all devices.
IOMMU	Auto	• **Enabled/Auto:** Enables IOMMU. AMD recommends setting this to `pt:pass-through` in the Linux kernel settings. • **Disabled:** Disables IOMMU.
AVIC	Disabled	Advanced Virtual Interrupt Controller. • **Disabled:** Disables AVIC. • **Enabled:** Enables AVIC.
x2AVIC	Disabled	x2AVIC is an extension of the advanced virtual interrupt controller. This feature currently requires a custom AMD Linux kernel. • **Disabled:** Disables x2AVIC. • **Enabled:** Enables x2AVIC.

Table 3-5: PCIe, I/O, security, and virtualization BIOS settings

TSME	Auto	• **Auto/Disabled:** Disables transparent secure memory encryption. • **Enabled:** Enables transparent secure memory encryption.
SEV	Disabled	In a multi-tenant environment (such as a cloud), Secure Encrypted Virtualization (SEV) mode isolates virtual machines from each other and from the hypervisor. • **Disabled:** SEV is disabled. • **Enabled:** SEV is enabled.
SEV-ES	Disabled	Secure Encrypted Virtualization-Encrypted State (SEV-ES) mode extends SEV protection to the contents of the CPU registers by encrypting them when a virtual machine stops running. Combining SEV and SEV-ES can reduce the attack surface of a VM by helping protect the confidentiality of data in memory. • **Disabled:** SEV-ES is disabled. • **Enabled:** SEV-ES is enabled.
SEV-SNP	Disabled	Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) mode builds on SEV and SEV-ES by adding strong memory integrity protection to create an isolated execution environment that helps prevent malicious hypervisor-based attacks such as data replay and memory re-mapping. SEV-SNP also introduces several additional optional security enhancements that support additional VM use models, offer stronger protection around interrupt behavior, and increase protection against recently-disclosed side channel attacks. • **Disabled:** SEV-SNP is disabled. • **Enabled:** SEV-SNP is enabled.

Table 3-5: PCIe, I/O, security, and virtualization BIOS settings

3.6 Higher-Level Settings

The system powers on to an initial state, after which succeeding software layers may affect system settings:

1. System firmware validates basic hardware functionality and allows users to change various settings via the BIOS Setup menus.
2. UEFI provides a shell environment that allows users to further interact with the system.
3. The operating system or hypervisor is the next software layer that provides control over system hardware.
4. Lastly, certain applications can also affect underlying hardware.

Each of the lines above may alter settings made by prior line, and some user changes require a reboot to take effect.

Please refer to your OEM documentation and/or applicable AMD Tuning Guide(s) for further guidance.

Chapter 4 Best Practices for Container Deployment

This chapter recommends best practices for container deployment on Kubernetes

4.1 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.

4.2 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.

4.2.1 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 (n2d-standard-2 master nodes):** 8 GiB of memory, 2 vCPU.
  - **500 nodes (n2d-standard-64 worker nodes):** 256 GiB of memory, 64 vCPU.
- **AWS:**
  - **5 nodes (m6a.large master nodes):** 8 GiB of memory, 2 vCPU.
  - **500 nodes (c6a.8xlarge worker nodes):** 128 GiB of memory, 64 vCPU.

4.2.2 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*.

4.3 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 4-1 describes the test environment.

4.3.1 Software Configuration

Table 4-1 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 4-1: 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 4-1: Software configuration (Continued)

4.3.2 Test Methodology

The tuning knobs suggested in "General" on page 21 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 5 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.

5.1 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.

For AMD EPYC 9xx4 processors (1P):

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-7,96-103 \
$KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARG
```

For AMD EPYC 97xx processors (1P):

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-7,128-135 \
$KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARG
```

Restart the kubelet service.

```
# systemctl restart kubelet
```

5.2 Container Pinning Settings

5.2.1 Enable the Static CPU Manager Policy

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-policy=static and - - reserved-cpus` to the `ExecStart` parameter.

For AMD EPYC 9xx4 processors:

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-7,96-103 \
--cpu-manager-policy=static \
$KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS $KUBELET_EXTRA_ARGS
```

Setting `-reserved-cpus` is a requirement to enable the Static CPU Manager. The indicated CPUs will not be allocatable to pods and are dedicated for system and kubelet daemons.

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

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

Restart the kubelet service. Only Guaranteed QoS pods will be pinned to CPUs when the Static CPU Manager Policy is enabled.

5.2.2 Lowest Level Cache Affinity using the Static CPU Manager Policy

Some workloads may see a performance boost when affinitized to the cores of the lowest level cache. Pods can be affinitized to a L3 cache grouping of cores by utilizing the Static CPU Manager Policy. The Static CPU Manager will assign Guaranteed QoS pods their cores in numerical order respective to the order which the pods are deployed and the user specified CPU resources for the pod. Figure 5-1 shows an example of how cores will be scheduled to guaranteed pods when the static policy is enabled.

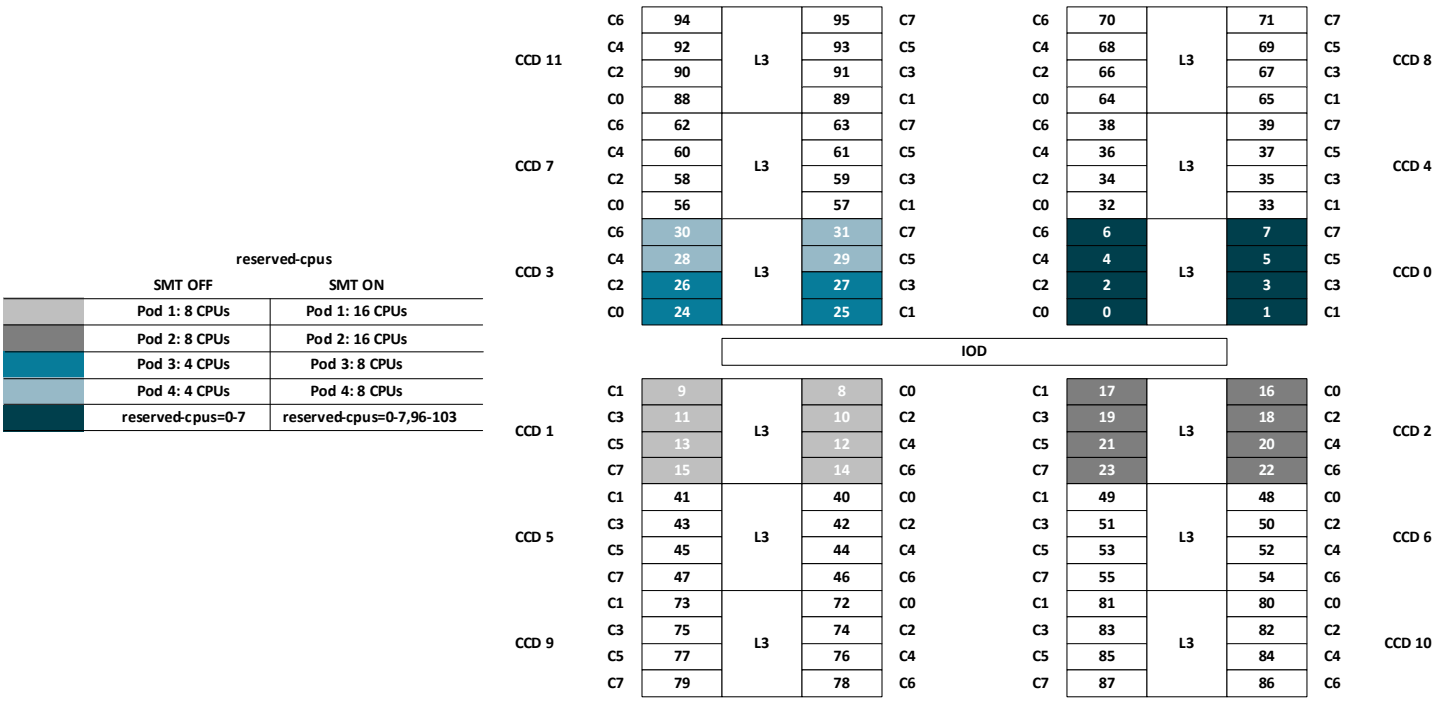

Figure 5-1: Scheduling cores to pods with the static policy enabled

The Static CPU Manager is SMT-aware. Static Guaranteed QoS pods will take precedence over Burtsable and Best Effort QoS pods. Burstable and Best Effort QoS pods can be deployed concurrently with Guaranteed QoS pods and will not impact the numerical core scheduling of static Guaranteed QoS pods.

5.3 Resolving the Noisy Neighbor Problem

5.3.1 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
```

5.3.2 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.

For AMD EPYC 9xx4 processors (1P):

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-7,96-103 \
--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.

5.4 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 29.

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

For AMD EPYC 9xx4 processors (1P):

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-7,96-103 \
--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
```

5.5 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 30 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.

For AMD EPYC 9xx4 processors (1P):

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-7,96-103 \
--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
```

5.6 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.

For AMD EPYC 9xx4 processors (1P):

```
ExecStart=/usr/bin/kubelet --node-ip=192.168.1.200 \
--reserved-cpus=0-7,96-103 \
--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
```

5.7 SR-IOV Network Device Plugin for High-Perf. Network I/O

Single Root I/O Virtualization (SR-IOV) allows a physical PCIe device to present itself multiple times as a virtual instance of the device. This is called Virtual Functions (VFs). A device plugin can be deployed as a Kubernetes DaemonSet to allow the node to recognize and assign particular VFs available on the host node to pods. Applications that require high bandwidth and low latency can see performance and reliability improvements by allocating dedicated VFs to its respective pod. The implementation described here used a Broadcom P2100G PCIe Network Interface Card (NIC), but any SR-IOV capable Ethernet NIC will work as well.

To install:

1. Ensure that SR-IOV is enabled in BIOS.

2. Identify the Physical Function (PF) name of the port of the SR-IOV-enabled card, and then run the following command to create 64 VFs:

```
# echo 64 > /sys/class/net/${PF_NAME}/device/sriov_numvfs
```

3. You must install a SR-IOV CNI plugin Kubernetes node to allow configuring and using VF networks in Kubernetes. This plugin requires Go 1.17+ to build. Clone the SR-IOV CNI plugin repository and run `make` inside the cloned repository to build the plugin binary:

```
# git clone https://github.com/k8snetworkplumbingwg/sriov-cni
```

   Upon build, the plugin binary will be located in `build/sriov`.

4. Move the binary to `/opt/cni/bin`.

5. Begin deploying the SR-IOV device plugin as a Kubernetes DaemonSet by building the device plugin image on each Kubernetes node. Do this by executing the following commands:

```
# git clone https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin
```

```
# make image
```

6. You will need to define the SR-IOV resource pool on each node to create device plugin endpoints based on your resource configuration. Here is an example of a `SRIOVConfigMap.yaml` file for the two ports of the Broadcom P2100G NIC:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sriovdp-config
  namespace: kube-system
data:
  config.json: |
    {
        "resourceList": [{
                "resourceName": "broadcom_sriov_netdevice_portA",
                "resourcePrefix": "broadcom.com",
                "selectors": {
                    "rootDevices": ["0000:81:00.0"]
                }
            },
            {
                "resourceName": "broadcom_sriov_netdevice_portB",
                "resourcePrefix": "broadcom.com",
                "selectors": {
                    "rootDevices": ["0000:81:00.1"]
                }
            }
        ]
    }
```

You can customize the `resourceName` and `resourcePrefix` fields to identify the node's PF. You must fill the `rootDevices` field with the corresponding SR-IOV-enabled NIC PCI address.

7. Execute the following command to deploy the configuration map:

```
# kubectl apply -f SRIOVConfigMap.yaml
```

8. You must also create a Network Attachment Definition in order to expose and request an SR-IOV interface in a container. Here is an example of a `NetworkAttachmentDefinition-Port-A.yaml` file for a single port of the Broadcom P2100G:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-net-a
  annotations:
    k8s.v1.cni.cncf.io/resourceName: broadcom.com/broadcom_sriov_netdevice_portA
spec:
  config: '{
  "type": "sriov",
  "ipam": {
    "type": "host-local",
    "subnet": "11.11.1.0/24",
    "rangeStart": "11.11.1.10",
    "rangeEnd": "11.11.1.73",
    "routes": [{
      "dst": "0.0.0.0/0"
    }],
```

```
    "gateway": "11.11.1.1"
  }
}'
```

You can customize the `name` field of the Network Attachment Definition to identify the network of a single NIC port. Populate the `k8s.v1.cni.io/resourceName` field with the `resourcePrefix/resourceName` fields that were provided in the previous Configuration Map. The `rangeStart` and `rangeEnd` fields need to provide a range of addresses that equal the required number of SR-IOV VFs. Each port of the SR-IOV-enabled NIC requires a Network Attachment Definition.

9. Execute the following command to apply the Network Attachment Definition:

```
# kubectl apply -f NetworkAttachmentDefinition-Port-A.yaml
```

10. Having defined the node's networking resources, deploy the SR-IOV Device Plugin DaemonSet using the following command inside the SR-IOV Network Device Plugin repository:

```
# kubectl create -f sriov-network-device-plugin/deployments/k8s-v1.16/sriovdp-
daemonset.yaml
```

11. A compatible CNI meta-plugin is required to enable the capability to attach the VF network interfaces to a pod. This implementation used Multus CNI was utilized, which can be installed by executing the following commands:

```
# git clone https://github.com/k8snetworkplumbingwg/multus-cni
# kubectl apply -f multus-cni/deployments/multus-daemonset-thick.yml
```

12. Upon a successful installation, the allocatable resource list for the node should discover the VF resources provided by the device plugin. Check this by executing the following command:

```
# kubectl get node ${Node_Name} -o json | jq '.status.allocatable'
```

In order to attach a discovered VF to a pod, ensure the pod asks for the required amount of VF resources from a specified device pool using the following format:

```
 apiVersion: v1
kind: Pod
metadata:
  name: testpod1
  labels:
    env: test
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-net-a
spec:
  containers:
  - name: iperf
    image: docker.io/library/iperf:latest
    ports:
    - containerPort: 2000
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "iperf3 -s -p 2000" ]
    resources:
      requests:
        broadcom.com/broadcom_sriov_netdevice_portA: '1'
        cpu: "1"
        memory: "512Mi"
      limits:
```

```
        broadcom.com/broadcom_sriov_netdevice_portA: '1'
        cpu: "1"
        memory: "512Mi"
  restartPolicy: "Never"
```

The `k8s.v1.cni.cncf.io/networks` field will be populated with the `name` field specified in the Network Attachment Definition of the desired VF resource pool. In order to attach the VF to the pod, the pod .yaml file must specify resource request and limits using the corresponding `resourcePrefix` and `resourceName` fields as shown above. More than one VF can be assigned to the pod.

5.8 RDMA Device Plugin for High-Perf. Network I/O

Remote Direct Memory Access (RDMA) allows direct data transfer between systems without CPU intervention to greatly improve throughput while lowering latency. A RDMA device plugin can be deployed on your Kubernetes cluster to allow pods to run native RDMA applications on the InfiniBand fabric. The implementation described below is for any NVIDIA Connect-X adapter card and was tested using a Connect-X 7 InfiniBand adapter card.

To install:

1. Ensure that the MLNX_OFED driver is properly installed.

2. Pull the RDMA Device Plugin Docker image by executing the following command:
   `docker pull mellanox/k8s-rdma-shared-dev-plugin`

3. If you are using ContainerD as the container runtime for the cluster, then use Docker to save the container image and import to ContainerD by executing the following commands:
   `docker save -o k8s-rdma-shared-dev-plugin.tar mellanox/k8s-rdma-shared-dev-plugin`

   `sudo ctr -n=k8s.io images import k8s-rdma-shared-dev-plugin.tar`

4. You will need to define the RDMA resource pool on each node with your Connect-X adapter to create device plugin endpoints based on your resource configuration. Here is an example of a `k8s-rdma-shared-dev-plugin-config-map.yaml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: rdma-devices
  namespace: kube-system
data:
  config.json: |
    {
        "periodicUpdateInterval": 300,
        "configList": [{
             "resourceName": "hca_shared_devices",
             "rdmaHcaMax": 1000,
             "devices": ["ibp225s0"]
           }
        ]
    }
```

You can customize the `periodicUpdateInterval` in a time interval in seconds to update the resources according to available host devices in the event of changes. The default is 60 seconds if none is specified. The `resourceName` is a customizable field to allow you to identify your Connect-X adapter. The maximum number of RDMA resources that you want the device plugin to provide can be customized using the `rdmaHcaMax` field. The devices field will require the PF of your Connect-X adapter.

5. Execute the following command to apply to device plugin configuration map:
   `kubectl create -f k8s-rdma-shared-dev-plugin-config-map.yaml`

6. Create the DaemonSet for the RDMA device plugin. Here is an example of a `k8s-rdma-shared-dev-plugin-ds.yaml` when using ContainerD as the container runtime:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: rdma-shared-dp-ds
  namespace: kube-system
spec:
  selector:
    matchLabels:
      name: rdma-shared-dp-ds
  template:
    metadata:
      labels:
        name: rdma-shared-dp-ds
    spec:
      hostNetwork: true
      priorityClassName: system-node-critical
      containers:
      - image: docker.io/mellanox/k8s-rdma-shared-dev-plugin:latest
        name: k8s-rdma-shared-dp-ds
        imagePullPolicy: IfNotPresent
        securityContext:
          privileged: true
        volumeMounts:
          - name: device-plugin
            mountPath: /var/lib/kubelet/
          - name: config
            mountPath: /k8s-rdma-shared-dev-plugin
          - name: devs
            mountPath: /dev/
      volumes:
        - name: device-plugin
          hostPath:
            path: /var/lib/kubelet/
        - name: config
          configMap:
            name: rdma-devices
            items:
            - key: config.json
              path: config.json
        - name: devs
          hostPath:
            path: /dev/
```

7. Execute the following command to apply the DaemonSet:
   `kubectl create -f k8s-rdma-shared-dev-plugin-ds.yaml`

   The cluster is now able to assign RDMA devices to pods. Here is an example of a test-hca-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: mofed-test-pod
spec:
  restartPolicy: OnFailure
  containers:
  - image: mellanox/rping-test
```

```
    name: mofed-test-ctr
    securityContext:
      capabilities:
        add: [ "IPC_LOCK" ]
    resources:
      limits:
        rdma/hca_shared_devices: 1
    command:
    - sh
    - -c
    - |
      ls -l /dev/infiniband /sys/class/infiniband /sys/class/net
      sleep 1000000
```

- You can assign more than one RDMA HCA device to a pod.

8. Execute the following command to run the pod:
   `kubectl create -f test-hca-pod.yaml`

This page intentionally left blank.

Chapter 6

Resources

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

This page intentionally left blank.

Chapter 7 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.

Chapter 8

Processor Identification

Figure 8-1 shows the processor naming convention for AMD EPYC 9004 Series Processors and how to use this convention to identify particular processors models:

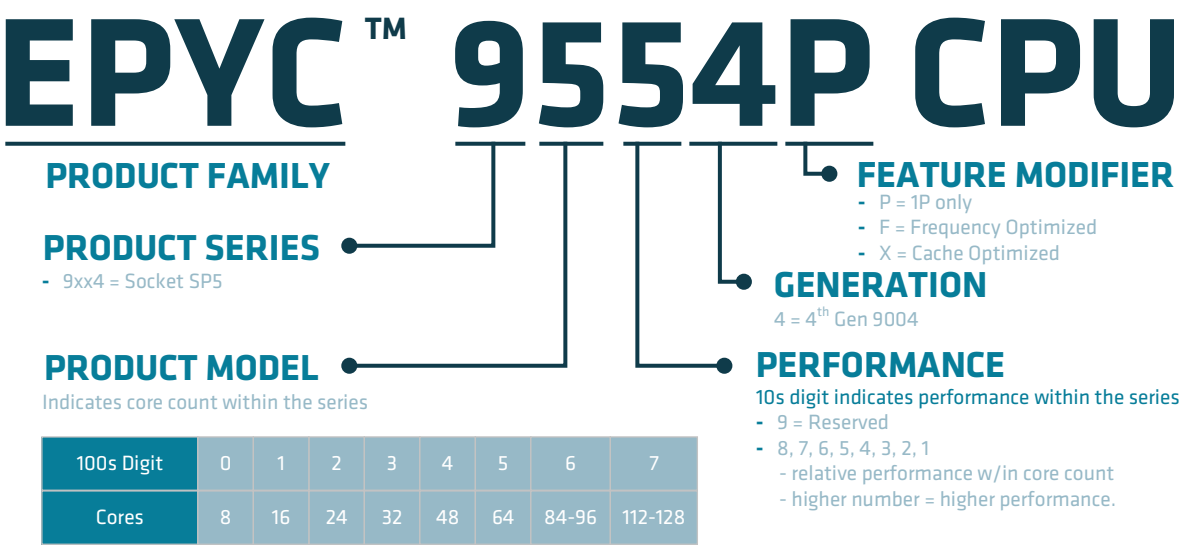

Figure 8-1: AMD EPYC SoC naming convention

8.1 CPUID Instruction

Software uses the `CPUID` instruction (`Fn0000_0001_EAX`) to identify the processor and will return the following values:

- **Family:** 19h identifies the "Zen 4" architecture
- **Model:** Varies with product. For example, EPYC Family 19h, Model 10h corresponds to an "A" part "Zen 4" CPU.
  - **91xx-96xx (including "X" OPNs):** Family 19h, Model 10-1F
  - **97xx:** Family 19h, Model A0-AF
- **Stepping:** May be used to further identify minor design changes

For example, `CPUID` values for Family, Model, and Stepping (decimal) of 25, 17, 1 correspond to a "B1" part "Zen 4" CPU.

8.2 New Software-Visible Features

AMD EPYC 9004 Series Processors introduce several new features that enhance performance, ISA updates, provide additional security features, and improve system reliability and availability. Some of the new features include:

- 5-level Paging
- AVX-512 instructions on a 256-byte datapath, including BFLOAT16 and VNNI support.
- Fast Short Rep STOSB and Rep CMPSB

Not all operating systems or hypervisors support all features. Please refer to your OS or hypervisor documentation for specific releases to identify support for these features.

Please also see the latest version of the *AMD64 Architecture Programmer's Manuals* or *Processor Programming Reference (PPR) for AMD Family 19h*.

8.2.1 AVX-512

AVX-512 is a set of individual instructions supporting 512-bit register-width data (i.e., single instruction, multiple data [SIMD]) operations. AMD EPYC 9004 Series Processors implement AVX 512 by "double-pumping" 256-bit-wide registers. AMD's AVX-512 design uses the same 256-bit data path that exists throughout the Zen4 core and enables the two parts to execute on sequential clock cycles. This means that running AVX-512 instructions on AMD EPYC 9004 Series will cause neither drops on effective frequencies nor increased power consumption. On the contrary, many workloads run more energy-efficiently on AVX-512 than on AVX-256P.

Other AVX-512 support includes:

- Vectorized Neural Network Instruction (VNNI) instructions that are used in deep learning models and accelerate neural network inferences by providing hardware support for convolution operations.
- Brain Floating Point 16-bit (BFLOAT16) numeric format. This format is used in Machine Learning applications that require high performance but must also conserve memory and bandwidth. BFLOAT16 support doubles the number of SIMD operands over 32-bit single precision FP, allowing twice the amount of data to be processed using the same memory bandwidth. BFLOAT16 values mantissa dynamic range at the expense of one radix point.