TUNING GUIDE

AMD EPYC™ 9005

Java®
Tuning Guide

AMD
together we advance_data center computing

PID: 58481
v1.0
October 2024

Anil Rajput, Gnanakumar Rajaram, and the AMD India Java team

DATE	VERSION	CHANGES
June, 2024	0.1	Initial NDA release
October, 2024	1.0	Initial public release

AUDIENCE

This document is intended for a technical audience such as Java® application architects, 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 both the BMC and OS-specific configuration, monitoring, and troubleshooting tools.

JAVA®
TUNING GUIDE
CONTENTS

Chapter 1: Introduction

This tuning guide provides detailed descriptions of parameters that can optimize Java® application performance on servers powered by AMD EPYC™ 9005 Series Processors. Default hardware and BIOS settings from different OEM vendors may require additional tuning for optimal performance depending on your workload(s). This guide helps you optimize Java on a per-platform and per-workload level by calling out:

- Hardware configuration best practices
  - Memory configuration
  - PCI subsystem
  - Recommended BIOS settings that can impact performance
- Software configuration best practices
  - OS configuration and kernel parameters
  - Java application command line options and best deployment practices

Production Java deployments often have complex architectures consisting of monolithic instances to microservices, containers, and call chain stack depths in the hundreds supported by an underlying web of hardware resources including systems, networking, and storage. The tunings and optimizations recommended in this tuning guide cannot solve complex bottlenecks but can enhance the performance of reasonably-operating Java deployments. Further, this tuning guide is not intended as a validation guide or as a generic server optimization guide.

This tuning guide provides information about JDK, OS, deployments, and available options for optimal hardware configuration. It does not cover software architecture bottlenecks, debugging, data flow decisions, etc. because specialized information is already publicly available. Java productions are very complex and often involve proprietary data that makes replicating production deployments almost impossible. Most of the recommendations in this guide are therefore based on analyses based on using the SPECjbb®2015 Java benchmark.

1.1 - Important Reading

Please be sure to read the following guides (available from the AMD Documentation Hub), which contain important foundational information about 5th Gen AMD EPYC processors:

- *AMD EPYC™ 9005 Processor Architecture Overview*
- *BIOS & Workload Tuning Guide for AMD EPYC™ 9005 Series Processors*
- *Memory Population Guidelines for AMD EPYC™ 9005 Series Processors*

1.2 - Java Platforms

Java is a widely accepted language used for many data-oriented applications that is popular for its simple, secure, robust, interpreted, and multithreaded features. Tens of thousands of enterprise applications are powered by Java and millions of people use them daily. From its inception, Java was designed to provide the flexibility for developers to code with features including architecture neutrality, automatic garbage collection, object orientation, and inheritance.

Java is among the top choices for segments such as mobile and desktop computing, gaming, etc., but this tuning guide focuses on server-side Java deployments. Performance becomes the key factor once your tested and debugged Java server application is ready for deployment. A balanced platform with optimal tunings should deliver expected performance gains over a reference platform. However, new platforms may have many changes at all levels such as BIOS, cores, NUMA, memory population, OS, Java Virtual Machines (JVM), and even application itself. This many changes make it challenging to identify optimal configurations for a given platform. Java benchmarks such as SPECjbb2015 from the Standard Performance Evaluation Corporation (SPEC) can help assess basic platform capabilities as well as provide valuable guidance on considerations such as instance sizing, scaling up and scaling out, and memory allocation. The SPECjbb2015 is extensively used to showcase performance and perform validation, testing, and tunings for both hardware and software.

This guide uses SPECjbb2015 to ensure optimal platform configuration and to showcase the impact of various platforms feature using the following SPECjbb2015 metrics:

- max-jOPS, which represents the maximum system sustained throughput.
- critical-jOPS, which is a Service Level Agreement (SLA) threshold-bound throughput representing typical production deployments.

SPECjbb2015 scales with compute and memory infrastructure while exercising reasonable network I/O throughput among Java instances with minimal disk I/O for log storage. It can mimic many scale-up and scale-out scenarios to understand the impact of Java production deployments using bare metal servers, hypervisors, containers, cloud VMs, etc.

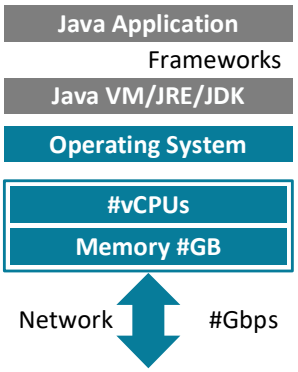

Figure 1-1: Java platform stack and deployments

Chapter 2: Hardware Configuration Best Practices

2.1 - BIOS Settings

Several BIOS settings can be tuned to improve performance. Evaluate the options shown in Table 2-1 to determine their impact on your targeted Java application. Please also see the *BIOS and Workload Tuning Guide for AMD EPYC™ 9005 Series Processors* (available from the AMD Documentation Hub). Default BIOS options generally produce the best overall performance for typical workloads. However, individual workloads differ and default options may not be best for all situations.

2.1.1 - Recommended Common BIOS Settings

Name	Value	Description
Local APIC Mode	x2APIC	Enabling x2APIC supports scaling beyond 255 hardware threads but is not supported by some legacy OS versions. Interrupt delivery is faster when using x2APIC mode compared to legacy xAPIC mode. If your OS supports x2APIC mode, then enable it even for configurations with <256 logical processors.
SMT Control	Enable	Enabling Simultaneous Multi Threading (SMT) allows two hardware threads per core, which may boost performance. The benefit of this setting depends on the application. Enabling SMT in BIOS is recommended for Java deployments unless there are other specific reasons to disable it.
LLC as NUMA Domain	Disable	Enabling this option reports each L3 cache as a NUMA domain to the OS. This creates a number of NUMA nodes equal to the total number of L3 caches in the system. AMD recommends disabling this for Java deployments unless you can leverage a higher number of NUMA nodes by pinning Java VM processes.
Node per Socket (NPS)	NPS [0|1|2|4]	This setting enables a trade-off between minimizing local memory latency for NUMA-aware or highly parallelizable workloads vs. maximizing per-core memory bandwidth for non-NUMA-friendly workloads. The following NPS settings are available: • **NPS4:** Recommended for maximum aggregated throughput from four or more Java VM instances per socket and the ability to pin Java VMs to the correct NUMA nodes. • **NPS2:** When using two Java VM instance per socket • **NPS1:** When using one Java VM instance per socket or a mix. • **NPS0:** Interleave memory accesses across all channels in both sockets. This only applies to 2P systems and is not recommended.

Table 2-1: Recommended common BIOS settings

L1 and L2 Stream HW Prefetchers	Enable	AMD recommends enabling this option for most Java applications. SPECjbb2015 delivers better performance when both prefetchers are disabled, although most Java deployments will benefit by enabling prefetchers. Most Java applications only moderately utilize the available memory bandwidth, unlike SPECjbb2015, which stresses it close to the maximum capacity.
Determinism Control	Manual	Enables the **Determinism Slider** control.
Determinism Slider	Power	Setting this to **Power** ensures maximum performance levels for each CPU given its capabilities.

Table 2-1: Recommended common BIOS settings (Continued)

2.1.2 - Advanced BIOS Settings for Latency and Cross-Socket Traffic

Name	Value	Description
xGMI Link Max Speed	Auto/32 Gbps	Monolithic/single JVM fully utilizing a 2P system can benefit from setting 4-link xGMI speed set to maximum 32Gbps for better cross-socket traffic handling, if supported.
APB Disable (APBDIS)	Auto/0	Enable or disable Algorithm Performance Boost (APB). By default, the AMD Infinity Fabric selects between a full- and low-power fabric clock and memory clock based on usage. Latency may increase in some scenarios involving low- bandwidth, latency-sensitive traffic as the processor transitions from low to full power. Setting APBDIS to 1 (APB disabled) and specifying a fixed Infinity Fabric P-state of 0 forces the AMD Infinity Fabric and memory controllers into full-power mode and eliminates latency jitter. Setting a fixed AMD Infinity Fabric P-State of 1 on certain CPU OPNs and memory population options reduces both memory latency and memory bandwidth, which may benefit applications that are sensitive to memory latency.
DF P-State	P0	DfPstate index to set below when APBDIS [1]. • **Min Value:** 0 (default) • **Max Value:** 4 • **P0:** Highest-performing AMD Infinity Fabric P-state • **Pn:** Next-highest-performing AMD Infinity Fabric P-state

Table 2-2: Recommended advanced BIOS settings

2.2 - Memory Configuration and DIMMs Population

It is very important to ensure that your memory subsystem is properly configured because it plays a crucial role in throughput and response time. Systems based on AMD EPYC 9005 processors include up to twelve DDR5 memory channels on each CPU socket. A single-socket or dual-socket platform must therefore have all recommended memory channels populated for optimal performance. Other configurations where all channels are not populated will reduce performance. Please see the latest version of Memory Population Guidelines for AMD Family 1Ah Models 00h–0Fh and Models 10h–1Fh Socket SP5 Processors - Login required.

2.3 - Java I/O Intensive Workloads

2.3.1 - NUMA Affinity

The performance of Java I/O-intensive workloads improves by placing the workload on the same NUMA node that connects to the I/O Device. For example, for a networking intensive operation, place the workload on the socket that the Network Interface Card (NIC) connects to. Command tools, such as `lstopo(hwloc)` on Linux, can help determine the connectivity between PCI devices and sockets. `lstopo(hwloc)` can display how the OS sees NUMA nodes, caches, cores, and PCI devices.

2.3.2 - IOMMU

The Input–Output Memory Management Unit (IOMMU) connects a Direct Memory Access-capable (DMA- capable) I/O bus to the main memory. The IOMMU provides several benefits and is required when using x2APIC, which must be enabled when using > 255 processor threads. IOMMU allows operating systems and hypervisors to provide extended memory support and protection for DMA-capable I/O devices as well as device interrupt remapping. Virtualized environments benefit from enabling this option.

If you believe the IOMMU is degrading performance, then resolve this based on the OS:

- **Linux OS:** Enable IOMMU in BIOS and disable it via OS options (e.g., `iommu=pt` on the Linux kernel command line). AMD has contributed code into the Linux kernel for years. One area of code contribution focused on optimizing the IOMMU code for AMD EPYC processors, which can directly impact TCP/IP performance in both virtualized and bare metal environment. Disabling the IOMMU or setting it to passthrough mode may be needed for high-bandwidth NICs, such as 200Gbps Ethernet adapters.
- **Windows:** Disable IOMMU in BIOS. For Windows Server 2016, 2019, and 2022, please refer to the latest versions of both the *Microsoft® Windows® Server Tuning guide for AMD EPYC 9005 Series Processors* (available from the AMD Documentation Hub) and the AMD I/O Virtualization Technology (IOMMU) Specification (login required).

2.3.3 - Preferred I/O and PCIe Relaxed Ordering

3rd Gen and prior AMD EPYC processors support PCIe **Preferred I/O** and **Relaxed Ordering** settings that helped optimize network and disk I/O performance. 5th Gen AMD EPYC processors (9xx5 models) include architectural enhancements that deliver optimal network and disk I/O performance by default without the need for either of these features.

2.3.4 - Jumbo Frames

A jumbo frame is an Ethernet frame with a payload greater than the standard Maximum Transmission Unit (MTU) of 1,500 bytes. Enabling jumbo frames in network-intensive deployments may significantly reduce interrupts and system CPU utilization and improve performance.

THIS PAGE INTENTIONALLY LEFT BLANK.

Chapter 3: Java Performance Tuning and Best Practices

Java application performance in production environments depends on many factors such as application architecture, bottlenecks, oversubscription of the resources, network and storage I/O, memory capacity or bandwidth, OS, Java Development Kit (JDK), and parameters. It is also very important to configure hardware platforms to match the application architecture and allow it to scale under various load conditions to meet the deployment's Service Level Agreement (SLA) requirements.

AMD EPYC platforms deliver excellent performance across a wide range of applications. Achieving full Java application performance requires using recommended OS versions, JDK versions, and configuring your platform to match your application needs. Based on extensive SPECjbb2015 testing, the following sections provide guidance relevant to many use cases and can easily be applied to the platform under evaluation or in production. If the goal is to replicate a published AMD EPYC platforms SPECjbb2015 result, you can find official submissions by searching the SPECjbb2015 Results -- Query*. Per SPECjbb2015 run and reporting rules, each result contains all the setup and configuration information. The remaining sections in this guide outline software and Java VM configuration options that affect application performance including tuning the operating system and kernel parameters for optimizing Java application performance.

3.1 - OS Configuration and Kernel Parameters

AMD recommends using the latest OS version available. Several kernel parameters work well with Java applications. For example, a Java application running large number of threads may benefit from avoiding too many context switches. See "Additional Information" on page 19 for information about optimal parameters for SPECjbb2015 performance. I/O-intensive applications must have sufficient resources for sockets and file handles. Networking best practices must also be followed. AMD recommends using simple monitoring tools such as `vmstat` or `perfmon` to correlate system resource utilization against expected performance.

3.2 - System Configuration

Most production deployments have very complex system configurations depending on the platforms and applications. This section lists some of the OS settings that boost SPECjbb2015 performance. SPECjbb2015 stresses a system by increasing load in 1% increments of the expected High Bound load until the system reaches maximum sustained capacity while measuring various response times, including the 99th percentile response time. This section presents the system configurations based on the published results at SPECjbb2015 Benchmark Results* for systems powered by AMD EPYC 9005 Series Processors. Using these settings and adjusting for different environments may be appropriate for many typical Java deployments. Please also see "Additional Information" on page 19 for more parameter information. For example, some Linux distros may discontinue parameters associated with `kernel.sched`. Please see "Kernel Parameters" on page 22 for additional information about alternative parameters.

```
cpupower -c all frequency-set -g performance
tuned-adm profile throughput-performance
echo 10000 > /proc/sys/kernel/sched_cfs_bandwidth_slice_us
echo 0 > /proc/sys/kernel/sched_child_runs_first
echo 16000000 > /proc/sys/kernel/sched_latency_ns
```

```
echo 1000 > /proc/sys/kernel/sched_migration_cost_ns
echo 28000000 > /proc/sys/kernel/sched_min_granularity_ns
echo 9 > /proc/sys/kernel/sched_nr_migrate
echo 100 > /proc/sys/kernel/sched_rr_timeslice_ms
echo 1000000 > /proc/sys/kernel/sched_rt_period_us
echo 990000 > /proc/sys/kernel/sched_rt_runtime_us
echo 0 > /proc/sys/kernel/sched_schedstats
echo 1 > /proc/sys/kernel/sched_tunable_scaling
echo 50000000 > /proc/sys/kernel/sched_wakeup_granularity_ns
echo 3000 > /proc/sys/vm/dirty_expire_centisecs
echo 500 > /proc/sys/vm/dirty_writeback_centisecs
echo 40 > /proc/sys/vm/dirty_ratio
echo 10 > /proc/sys/vm/dirty_background_ratio
echo 10 > /proc/sys/vm/swappiness
echo 0 > /proc/sys/kernel/numa_balancing
echo always > /sys/kernel/mm/transparent_hugepage/defrag
echo always > /sys/kernel/mm/transparent_hugepage/enabled

# Add this line to GRUB_CMDLINE_LINUX_DEFAULT # in /etc/default/grub

cgroup_disable=memory,cpu,cpuacct,blkio,hugetlb,pids,cpuset,perf_even t,freezer,devices,net_cls,net_prio

# And generate new grub using:(this requires a reboot to take effect) grub2-mkconfig -o /boot/grub2/
grub.cfg

ulimit -n 1024000
echo UserTasksMax=970000 >> /etc/systemd/logind.conf
echo DefaultTasksMax=970000 >> /etc/systemd/system.conf
```

3.3 - Java Virtual Machine (JVM)

Java VM may play an important role in helping a Java deployment achieve optimal performance. New platforms often include new architecture features that must be enabled in the OpenJDK or associated libraries or frameworks. The AMD Java team works with the OpenJDK* community to enable new platform features and to enhance Java performance for system powered by AMD EPYC processors.

AMD has contributed optimizations for AMD EPYC processors in Java SE 10, Java SE 11 Long Term Support (LTS), Java SE 17 LTS, Java SE 21 LTS, and later upcoming versions such as Java SE 22. AMD EPYC processors deliver excellent performance for most production JDKs, such as Java SE 8, however AMD recommends using Java SE 11 LTS or later versions from OpenJDK or Oracle JDK for optimal performance. You can download the latest Java SE from publicly available builds or can use a custom build with AMD EPYC architecture-specific flags using available OpenJDK code repositories.

3.3.1 - Java SE Versions

The Java language and platform have undergone many changes since its inception in 1995. Most production deployments use Java Platform, Server Edition 8 Long Term Support (Java SE 8 LTS), with many moving to Java SE 11 LTS, which was released in September 2018. Upcoming Java SE versions often enable the newer Instruction Set Architecture (ISA) of each new generation of AMD EPYC processors.

Java Version	Release Date	Notes
JDK Beta	1995	
Java SE 7	Jul 2011	
Java SE 8 (LTS)	Mar 2014	Most production deployments
Java SE 9	Sep 2017	
Java SE 10	Mar 2018	

Table 3-1: Java SE versions

Java SE 11 (LTS)	Sep 2018	Production deployments early adopters
Java SE 12	Mar 2019	
Java SE 13	Sep 2019	
Java SE 14	Mar 2020	
Java SE 15	Sep 2020	
Java SE 16	Mar 2021	
Java SE 17 (LTS)	Sep 2021	LTS after JDK11 and the most leading-edge
Java SE 18	Mar 2022	
Java SE 19	Sep 2022	
Java SE 20	Mar 2023	
Java SE 21 (LTS)	Sep 2023	

Table 3-1: Java SE versions (Continued)

3.3.2 - Just-In-Time Compilers (JIT)

HotSpot JVM is part of the OpenJDK project and contains two JIT-compilers:

- **C1 (client) compiler:** Compiles the interpreted methods (after a threshold of "hotness") at the optimization levels, 1, 2, and 3.
- **C2 (server) compiler:** Compiles the methods at level 4 (the highest optimization level).

C1 and C2 run within the context of the application along with the application and compile the "hot" application methods at various compilation levels. This is called "tiered" compilation and helps optimize application runtime. The four optimization levels are:

- Level 1 is simple optimization.
- Level 2 optimizes with minimal profiles.
- Level 3 optimizes with the full profile.
- Level 4 performs full optimization using the full profiles collected at Level 3.

The compiler flags may be useful for tuning the Java server application. For example, you can compile any method at any level using the given options. You can also exclude compiling methods if there are issues in compiling those methods. You can also change the amount of inlining, loop optimization, etc., as applied by the compiler using different JVM flags. In general, AMD recommends not changing the compiler flow because Hotspot does an excellent job of compilation. However, you can provide specific hints to the compiler by tuning the application using the flags provided. In the SPECjbb2015 example, AMD found that inlining a few big methods at an early stage improves performance.

If Java process startup time is important, then use Ahead-Of-Time (AOT) compilation introduced in Java 9 (https://openjdk.java.net/jeps/295*) and add the `jaotc` command-line utility. AMD EPYC™ processors support and run AOT compilations, but specific tunings are beyond the scope of this tuning guide.

3.3.3 - JVM Compilation

The OpenJDK project supports many native compilers to build the JDK. AMD team is collaborating to update the build process to include optimization flags for 2nd and 5th Gen AMD EPYC processors. There was no noticeable performance improvement for SPECjbb2015 as kernel and native components are less than 5% of the total SPECjbb2015 profile. There may be some performance gain for the workloads with larger kernel and native components when JDK builds include specific flags for the 2nd and 5th Gen AMD EPYC processors.

3.4 - Java Instance Sizing (Scale Up) and Scaling (Scale Out)

Scale up and scale out decisions are critical for any Java production deployment. Evaluating scale up requires verifying that additional resources are delivering the expected performance gains. AMD recommends aligning an instance to CCDs or processor quadrants when scaling up. If scaling up is utilizing more than one socket, then both sockets should use a balanced vCPU allocation approach.

When scaling out, it is important for each Java instance to have enough vCPUs and associated memory from local channels. Significant gains in throughput and response time are possible if vCPUs and associated memory are fully leveraging NUMA. The following subsections discuss tuning recommendations for optimal scale up and scale out performance.

3.4.1 - Scale Up

The first step in any deployment is to determine the optimal scale up for a single Java process. This process is an iterative closed feedback loop variables such as vCPUs, available memory, network, and storage bandwidth, etc. are increased while observing throughput and response time. A platform may support scaling up to the full system, but the deployment may limit this for other reasons such as replication, single point of failure, etc.

Any application alignment to vCPUs, cache, and memory architecture can deliver significant performance gains and faster response times. 5th Gen AMD EPYC processors have CCDs that offer 16 vCPUs when SMT is enabled and 8 vCPUs when SMT is disabled. AMD recommends that a single Java process should use vCPUs either within one CCD or multiple CCDs while avoiding partial CCDs or being unbalanced across a dual- socket system. It is also important that the memory channels associated with those CCDs have sufficient memory for the Java process. For example, if a java process uses 2 CCDs with 32 vCPUs (SMT ON) and needs almost 64GB memory, then the two associated memory channels should be populated with 64GB of memory. Query the platform's available processors and memory to verify that optimal configurations needs are being met.

3.4.2 - Scale Out

Evaluating the scale out strategy is next after determining the optimal number of vCPUs and memory per Java instance. This involves approximating overall scale out performance, deciding whether to align and pin instances to CCDs and NUMA nodes, network configurations, system and Java parameters, etc. It is also very important to monitor CPU, memory, network, and other resources to identify and address bottlenecks and estimate expected performance during scale out.

Scaling out performance should be close to linear for cloud deployments where each additional Java instance will likely use a different node. On-premises systems should scale well unless network bandwidth or other shared resources become contended or bandwidth-constrained. You can identify this by monitoring platform resources and addressing them by following various deployment rules and strategies.

3.5 - Java Deployment Decisions and NUMA Settings

NUMA architecture is prevalent in most datacenter and cloud servers. Production deployments may restrict pinning processes and memory to specific vCPUs or NUMA nodes. NUMA assigns local memory to each core, which can also access memory assigned to other cores. NUMA is called "non-uniform" because memory access times are faster when a processor accesses its own memory instead of borrowing memory from another processor. AMD EPYC processors also provide many NUMA configurations that you can leverage for optimal performance based on application instance vCPU and memory requirements. From a deployment perspective:

1. Evaluate the number of NUMA nodes needed for optimal performance. vCPUs and memory allocations for each Java application instance should be a good guide for this estimation provided that maximum scale out loads must be evaluated before production deployments.
2. Set the NPS setting, per "BIOS NPS Setting" on page 12 and the latest version of the *BIOS & Workload Tuning Guide for AMD EPYC™ 9005 Series Processors* (available from AMD EPYC Tuning Guides). If possible, decide whether to allow application instances to float across the whole platform or leverage NUMA pinning to achieve optimal performance and response time.

Other deployment choices such as being able to do VM migration etc. may prohibit NUMA pinning. Consider these such needs along with Java vCPUs and memory sizes, and then bind the JVM process(es) to a set of NUMA nodes and memory in such a way as to keep memory accesses as local as possible to the associated vCPUs. NUMA pinning requires deciding whether to pin memory, vCPUs, or both. You can also pin vCPUs to specific CPU threads, CCXs, CCDs, or NUMA node level. Be aware that when pinning across specific vCPUs, mapping of vCPUs to physical cores and CPU threads can vary across various Operating Systems and it is very different between Windows OS and Linux.

3.5.1 - Optimal CCX/CCD Alignment

AMD EPYC 9005 Series Processors have up to sixteen CCDs in each socket (1P) while some OPNs may have different numbers of CCDs. Each CCD shares an L3 (last level) cache across all cores in that CCD. A process runs optimally when it accesses memory from memory channels local to the CCDs. This requires pinning a Java VM to the CCDs. Figure 3-1 shows several examples of optimal alignments to different numbers of CCDs as well as examples of sub-optimal alignments either across CCDs or across sockets on a dual-socket system.

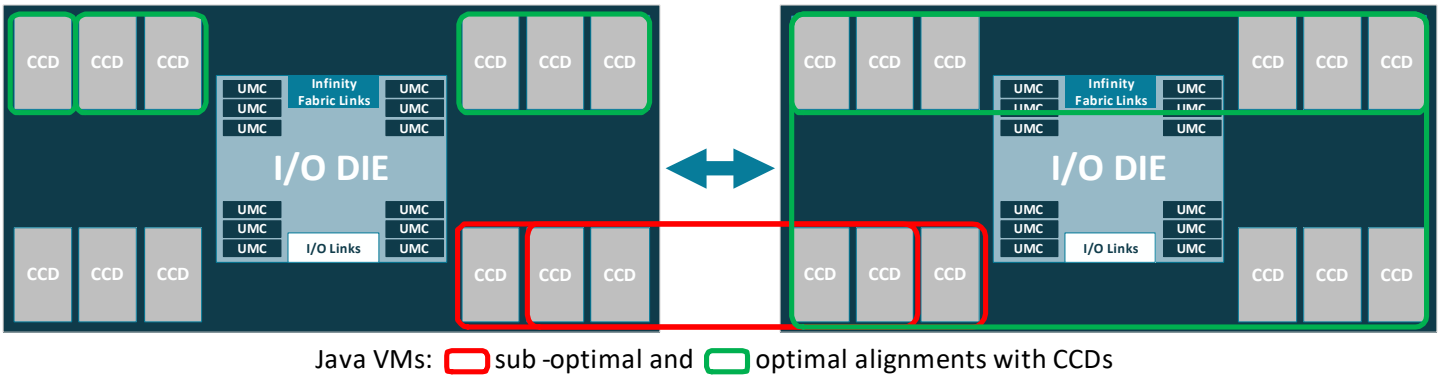

Figure 3-1: Java VM alignment with CCDs on a dual-socket AMD EPYC 9005 Series Processor platform with 12 CCDs/socket

Using a dual-socket system with 192 physical and 384 logical processor cores as an example, having either one Java instance per CCD (8C/16T) or 2 CCDs in same quadrant(16C/32T) will perform better than having one Java instance with 8C/16C split between 2 CCDs from different quadrants or sockets. AMD also recommends keeping Java VM VCPU size within either one NUMA node or a multiple of NUMA nodes.

3.5.2 - BIOS NPS Setting

AMD EPYC 9005 Series Processors provide Node Per Socket (NPS) settings of NPS1, NPS2, and NPS4 that create 1, 2, and 4 NUMA nodes per socket, respectively. You can determine the optimal NPS setting for AMD EPYC 9005 Series Processors based on the number of vCPUs per Java VM instance match as closely as possible to the vCPUs in a NUMA node or that fit best within a NUMA node.

3.5.3 - LLC as NUMA Domain

Disabling LLC as NUMA means that the NPS setting will determine the number of NUMA nodes and processor threads in each node. This will create similar results in AMD EPYC 9005 and earlier processors. Enabling LLC as NUMA creates NUMA nodes associated to each CCX (LLC) while associating memory channels to these NUMA nodes is still set by the NPS setting. Be sure to evaluate and update your deployment scripts to leverage this in AMD EPYC 9005 Series Processors.

1P System AMD EPYC 9005 Series Processor with 8 CCDs	#NUMA Nodes when LLC as NUMA is:		Memory Interleaving when LLC as NUMA (Enabled or Disabled)
	Enabled	Disabled	
NPS1	# of CCXs	1	Across all the channels in a socket.
NPS2	# of CCXs	2	All the channels are divided in two groups. Across the channels in a group provided these channels have DIMM populated in them.
NPS4	# of CCXs	4	All the channels are divided in four groups. Across the channels in a group provided these channels have DIMM populated in them.

Table 3-2: #NUMA nodes and memory interleaving for a single AMD EPYC 9005 Series Processor

2P System AMD EPYC 9005 Series Processor with 8 CCDs	#NUMA Nodes when LLC as NUMA is:		Memory Interleaving when LLC as NUMA (Enabled or Disabled)
	Enabled	Disabled	
NPS0	2x # of CCXs	1	Across all channels in both sockets. This is not recommended.
NPS1	2x # of CCXs	2	Across all the channels in each socket.
NPS2	2x # of CCXs	4	All the channels are divided in two groups in each socket. Across the channels in a group provided these channels have DIMM populated in them.
NPS4	2x # of CCXs	8	All the channels are divided in four groups in each socket. Across the channels in a group provided these channels have DIMM populated in them.

Table 3-3: #NUMA nodes and memory interleaving for dual AMD EPYC 9005 Series Processors

3.5.4 - Unpinned Java Deployments

As additional Java instances are deployed on the same system:

- The ability to pin to a given NUMA node and align within CCD boundaries provides an important mechanism for resource isolation and optimal performance as additional Java instances are deployed on the system. Otherwise, the vCPUs assigned to a VM can be allocated to any CPU cores, which may cause increased memory latency and a degraded user experience.
- Many production deployments cannot leverage NUMA bindings. Setting NPS1 delivers somewhat suboptimal performance but does deliver a very consistent user experience thanks to interleaving the memory accesses across all memory channels.

3.6 - Enabling Huge Pages

Most operating systems support huge pages. Using huge pages is both easy and can deliver up to 10% performance gain for many Java applications. Discuss this with your deployment team. Huge pages are locked in memory, which restricts memory oversubscriptions. Most Java VMs use 2M huge pages, but OpenJDK-based Java VMs can also use 1GB huge pages.

Windows allows large pages when enabled by an administrator and then using a Java parameter to tell the JVM to use huge pages. Linux uses two approaches to huge pages. The first approach requires allocating enough huge pages while the second approach uses transparent huge pages.

Enabling huge pages is a two-step process for bare-metal deployments at the OS and JVM levels. This includes an additional step for virtualization VMs, cloud VMs, or containers.

The OS-level enablement process varies by OS. Many recent Linux builds enable huge pages by default using a feature called Transparent Huge Pages. See our OS documentation for OS-specific instructions.

Most Java VMs also enable huge pages by default if it is already enabled by default at the OS, VM, or container level. Refer to the Java VM documentation for instructions about enabling huge pages. Deployments that use virtualization or cloud VMs or containers must have huge pages enabled at the VM and container level. These configurations are beyond the scope of this tuning guide.

Here is an example of enabling 2M/1G Hugepages and verifying that 128x 1GB hugepages are evenly distributed across all NUMA nodes.

```
echo 128 > /sys/devices/system/node/node$N/hugepages/hugepages- 1048576kB/nr_hugepages
echo 8192 > /sys/devices/system/node/node$N/hugepages/hugepages- 2048kB/nr_hugepages
cat /sys/devices/system/node/node$N/hugepages/hugepages- 2048kB/nr_hugepages
cat /sys/devices/system/node/node$N/hugepages/hugepages- 1048576kB/nr_hugepages
cat /sys/devices/system/node/node*/meminfo |grep HugePages_Free
```

Huge pages often provide a significant performance gain but may also increase performance variability because some Java VMs may allocate huge pages while others may not. Transparent Huge Pages are given to a requesting Java VM if the memory allocator has huge pages available, else it will allocate regular pages. It is not easy to detect when huge pages are being used unless the Java log specifically prints this out or if monitor huge pages use at the system level. You can specifically allocate huge pages to avoid performance variations. You can opt to modify Java VM options such as to have Java VM throw an error if it cannot allocate huge pages.

Java applications that access large data in memory are more likely to benefit from huge pages. Applications with code footprints greater than 2MB can also leverage huge pages for code that has similar processes. Again, the decision about whether or not to enable huge pages in a production environment requires a serious evaluation of the pros and cons.

3.7 - Java Parameters

Java parameters are very important for most deployments. Each Java SE must be checked when new parameters are introduced and current parameters are deprecated. Many default parameters may also change from one Java SE to the next. For example, Java SE 8 has old parallel GC as default type GC, while Java SE 11 made G1GC the default GC, which significantly changed performance and response times.

`$ java [-server] -XX:+PrintFlagsFinal -version` lists all flags to select for tuning Java applications for:

- Memory options
- Garbage collection options
- Compiling, interpreting, and other options

The following types of JVM flags are available:

- **Boolean:** Can be turned on with -XX:+ and off with -XX:-.
- **Numeric:** Can be set with -XX. Numbers can include 'm' or 'M' for megabytes, 'k' or 'K' for kilobytes, or 'g' or 'G' for gigabytes. For example, 32k is the same as 32768.
- **String:** Can be set by using -XX and is usually used to specify a file, a path, or a list of commands.

3.7.1 - Java Parameters to Optimally Leverage 5th Gen AMD EPYC Architecture

There are hundreds of Java parameters for different types of deployments. This section suggests a unique combination of Java parameters that may provide an additional performance boost when using systems powered by AMD EPYC 9005 Series Processors.

Based on SPECjbb2015 testing, the following combination should positively affect performance for a wide array of applications:

```
"-XX:+UseUnalignedLoadStores
-XX:+UseXMMForArrayCopy
-XX:+UseXMMForObjInit
-XX:+UseFPUForSpilling
-XX:-UseFastStosb"
```

AMD EPYC 9005 Series Processors support fast string operations, which you can enabled or disabled in BIOS. Enabling fast string operations enables the `UseFastStosb` option by default for object initialization. AMD testing showed that XMM/YMM MOVDQU instructions perform better overall, especially with respect to object sizes > 64 bytes & < 1KB. Generate XMM/YMM `movdqu` instructions for object initialization by using a combination of the following three parameters: `-XX:-UseFastStosb - XX:+UseUnalignedLoadStores -XX:+UseXMMForObjInit`.

Java heap object density and alignment are important for many applications. The Java parameter combination `-XX:+UseCompressedOops -XX:ObjectAlignmentInBytes` can deliver an additional performance gain when adjusted for the Java heap size. For example:

- **Java Heap up to 32GB:** -XX:+UseCompressedOops -XX:ObjectAlignmentInBytes=8 (default)
- **Java Heap up to 64GB:** -XX:+UseCompressedOops -XX:ObjectAlignmentInBytes=16
- **Java Heap up to 128GB:** -XX:+UseCompressedOops -XX:ObjectAlignmentInBytes=32

Different Java SE versions may have different default values for these parameters. AMD recommends explicitly define these parameters for Java deployments.

3.7.2 - Java Heap Memory Allocation Best Practices

Java heap memory allocation is a very complex topic because it requires evaluating many deployment parameters while meeting application needs within available resource constraints. All Java processes should always be launched with optimal java heap parameters. At one end, too small a Java heap may throw OutOfMemoryError (OOM) errors. On the other end, too large a Java heap may cause insufficient available system memory.

Systems powered by AMD EPYC 9005 Series Processors can support up to 12 memory channels per socket, and a dual-socket system can support up to twice that. Typical production systems often have much smaller total system memory. Many virtualization VMs, hypervisor guest operating systems, Kubernetes nodes, containers, and cloud VMs may also have their own total available memory constraints. Be sure that Java heap memory can meet all deployment requirements. A typical Java deployment empirical formula is: Number of Java VMs x (Heap memory + ~500MB memory for other resources) + ~500MB memory for OS must be less than the total available system memory for these Java processes.

3.7.3 - Avoid Using -XX:+UseNUMA

JVM includes the `UseNUMA` option that lets JVM exploit the server memory NUMA behavior. Use this option with caution. This option is not recommended if you are already using NUMA at the Java process level for cores and memory.

It is not a good practice to use the `UseNUMA` flag with a single NUMA node. Java SE 11 and later automatically disable this flag if used on a single NUMA node. If you enable `UseNUMA` flag and do not bind the java application to any NUMA node, then JVM creates the number of Lgrps (Locality Groups) based on the default system NUMA nodes. If you are running a single java application on a multi-NUMA node system, then AMD advises enabling `UseNUMA` flag because it helps assign heap space according to local NUMA nodes.

If your JAVA workload/application has to use JDK versions prior to Java SE 11, then evaluate the performance impact of the `-XX:+UseNUMA` flag. Java SE 11 to Java SE 13 addressed a set of Java performance bugs with the `UseNUMA` flag.

3.7.4 - Using Thread-Level NUMA Affinity Within Java VM Processes

This scenario applies to very specific Java applications where different thread pools are working on mostly dedicated data sets. Performance improves if you can identify such threads using the thread name and you can pin these threads to specific NUMA nodes or specific CCDs in AMD EPYC processors. For example, when using the SPECjbb2015 composite category with multiple groups within one Java process, you can pin Java threads associated to each group to specific NUMA nodes or CCDs. This extreme optimization improves performance for SPECjbb2015 max-jOPS and critical-jOPS.

For example, in a Java process with two distinct modules named Group1 and Group2, you can use the code below to pin Group1 to processor threads 0-63 and 128-191 which map to socket 1, and Group2 to processor threads 64-127 and 192-255 which map to socket 2. Mapping processor threads to a socket depends on the OS (either Linux or Windows). You can execute the Linux command `$ cat /proc/cpuinfo` to decipher the processor threads to sockets mapping. In Windows, you can use Task Manager option.

```
GROUP1_THREADS=(`jstack $JAVA_PID | grep Group1 | awk -F'nid=' '{printf "%s\n",$2}' | gawk -F' ' '{printf
"%d ", strtonum($1)}'` )
for i in "${GROUP1_THREADS[@]}" do
taskset -pc 0-63,128-191 $i done
GROUP2_THREADS=(`jstack $JAVA_PID | grep Group2 | awk -F'nid=' '{printf "%s\n",$2}' | gawk -F' ' '{printf
"%d ", strtonum($1)}'` )
for i in "${GROUP2_THREADS[@]}" do
taskset -pc 64-127,192-255 $i done
```

3.8 - GC Tuning

See Java Garbage Collection Basics* for a complete description of Java garbage collection basics. This section presents a summary for quick reference.

Allocating as small a heap as possible helps the application run more smoothly. The JVM heap is mainly divided into two generations called young generation "young gen" and old generation "old gen". 90% of the objects in most Java server class applications die very young and can be collected during young gen garbage collection (GC).

The young generation can have an Eden Space and two survivor spaces called From and To. Survivor spaces contain the tenured objects. A tenured object that reaches a certain threshold (set by a parameter) is transferred to old gen. When the Eden space is full, the young gen GC recovers the memory over a very short period, and the application experiences a short pause that affects performance.

An object surviving for longer periods is kept in old gen. Full GC occurs when the whole heap is full and has no space for new objects but experiences a longer pause time to clean the full heap memory. Avoid these longer pauses by setting the gen size as low as possible.

GC tuning is typically performed by studying the GC profiles from the application. The JVM writes these profiles with the specific flag `-Xlog:gc` (JDK 9 and above). These profiles help you understand GC behavior, such as the amount of memory freed, the amount of time spent doing minor (Eden) and major (full) GCs, the amount of heap space wasted (unused), etc. The developer must optimally increase/decrease the sizes of the different heaps (e.g. Eden, Survivor space, Old Gen) to minimize the amount of GC time. "Additional Information" on page 19 provides the SPECjbb2015 parameters as a case study.

3.9 - Throughput vs. Response Time

Most Java deployments can be categorized using a throughput focus such as batch processing or a response time focus such as high-frequency trading, user interactions, business criticality, etc. An optimal combination of application architecture along with the right JVM parameters are required to meet the desired goals.

The number of threads allocated to the various thread pools in an application is one of the most important tuning factors affecting performance and response time. If the total number of threads created by an application are more than the number of available hardware threads, then the environment may need careful tuning. There may also be different types of threads, such as (but not limited to) producer, consumer, or network. The application architecture and data flow may make it best to tune the number of such threads for optimal performance and response time from a given platform. The current Linux OS also provides various priority orders for the available threads that a given Java VM may use for Java JIT, GC, and so forth. For example, SPECjbb2015 reports the two metrics described in "Java Platforms" on page 2.

Extensive SPECjbb2015 tuning using number of threads and other parameters yielded the following observations:

- Tuning focused on SPECjbb2015 used 5x or more available processors threads and stop the world parallel GC -XX:+UseParallelGC.
- Tuning focused on SPECjbb2015 critical-jOPS used 2x available processors threads and more consistent response by using concurrent garbage collections with pause time limits like G1GC etc.

3.10 - Orchestration and Container Settings

Production Java deployments often use orchestration involving various type of infrastructure, such as bare-metal, containers, VMs, and public, private and hybrid clouds. Orchestration frameworks like Kubernetes automate deployment, scaling, and management of containerized applications. An initial test setup created using Kubernetes for orchestration and Docker as a container for SPECjbb2015 has suboptimal performance settings related to container overlay, networking, file systems, and health monitoring. The following references provide additional information about network configurations and performance comparisons among popular orchestrators:

- Kubernetes Cluster Networking*
- Docker Networking Overview*

You can reduce overhead related to `containerd` by pinning it to certain cores. Docker, Kubernetes, and other container platforms use `containerd` to abstract away `syscalls` or OS-specific functionality to run containers on Linux, Windows, and other operating systems. Here is an example of pinning `containerd`:

```
$ sudo vim /usr/lib/systemd/system/containerd.service
[Service]
CPUAffinity=24
```

THIS PAGE INTENTIONALLY LEFT BLANK.

Chapter 4: Additional Information

Garbage Collection

You can evaluate Java application performance using many online GC analyzing tools that use GC logs. You can collect these logs by running the application along with the following flag: -Xlog:gc=info:file=filename.log. Please see Unified Logging Of JVM Messages With The -Xlog Option* for additional information about unified logging.

- Set the Heap flags `-Xmx` `-Xms` `-Xmn` (Maximum, Starting, and Young Gen heap sizes, respectively) according to a given application profile. It is very important that the total allocated memory by all of the Java processes on a given platform to not result in memory swap or OOM (Out of Memory). For example, the optimal SPECjbb2015 max-jOPS heap settings are `- Xmx28g` `-Xms28g` `-Xmn26g` ("`g`" is for GB).
- Set the number of GC threads based on equal access to available processors for that Java VM process. For example, a Java VM process associated to an AMD EPYC processor with 1 CCD (8C/16T) should have an optimal value of 16 GC threads.
- Use Survivor Ratio to set the size of Eden and the two survivor spaces. Calculate the survivor space value using the formula Single Survivor Space Size = Young Gen Size / (Survivor Ratio+2)
- Use Max Tenuring Threshold to set the young gen object survival age. For example, setting the threshold to 15 moves objects to old gen after 15 young gen GC.

Of the available GC's (`Parallel`, `G1GC`, and `ZGC`), Parallel GC delivers the best SPECjbb2015 max- jOPS and is the default GC for Java SE 8 or earlier. The default GC on Java SE 11 changed from Parallel to G1GC. Please see The java Command* for additional information.

4.1 - NUMA Binding

```
numactl --physcpubind=0,1,2,3,4,5,6,7 --membind=0 <JAVA Application>
```

Node Level binding:

```
numactl --cpunodebind=0 --membind=0 <JAVA Application>
```

- **--cpunodebind:** Binds to the given node(s)
- **--physcpubind:** Binds to physical cores. Check the core mapping before binding.
- **--interleave=all:** Interleaves the memory across all the NUMA nodes.
- **--membind=0:** Binds memory to given node(s). Ensure matching of CPUs and associated local memory.
- **--localalloc:** Allocates the memory locally according to the `cpunodebind`.

Do not change any other parameter that depends on the number of available processors according to the binding, such as the number of GC threads matching the number of available processors. Further, do not allocate more memory than is available on a given node. Execute the command

`numactl -H` to check the node mapping and available memory on a given node. Please see numactl(8) - Linux man page* for additional information.

4.2 - NUMA Binding of Java VM Processes for 2P AMD EPYC 9005 Series Processors

BIOS (for an 8 CCD, 64-core CPU)	LLC as NUMA Domain Disabled (NPS4)
Number of NUMA nodes shown by `# numactl -H` shows	• **NUMA node(s):** 8 • **NUMA node0 CPU(s):** 0-15,128-143 • **NUMA node1 CPU(s):** 16-31,144-159 • **NUMA node2 CPU(s):** 32-47,160-175 • **NUMA node3 CPU(s):** 48-63,176-191 • **NUMA node4 CPU(s):** 64-79,192-207 • **NUMA node5 CPU(s):** 80-95,208-223 • **NUMA node6 CPU(s):** 96-111,224-239 • **NUMA node7 CPU(s):** 112-127,240-255
Number of CCDs in a NUMA node	2
8 JVMs by binding each JVM to a NUMA node	`numactl --cpunodebind=$node --membind=$node`
16 JVMs by binding JVMs to a NUMA node and using CPU cores within a NUMA node. In this example, multiple JVMs run tied to a NUMA node.	Socket 0: numactl --physcpubind=0-7,128-135 --membind=0 numactl --physcpubind=8-15,136-143 --membind=0 numactl --physcpubind=16-23,144-151 --membind=1 numactl --physcpubind=24-31,152-159 --membind=1 numactl --physcpubind=32-39,160-167 --membind=2 numactl --physcpubind=40-47,168-175 --membind=2 numactl --physcpubind=48-55,176-183 --membind=3 numactl --physcpubind=56-63,184-191 --membind=3 Socket 1: numactl --physcpubind=64-71,192-199 --membind=4 numactl --physcpubind=72-79,200-207 --membind=4 numactl --physcpubind=80-87,208-215 --membind=5 numactl --physcpubind=88-95,216-223 --membind=5 numactl --physcpubind=96-103,224-231 --membind=6 numactl --physcpubind=104-111,232-239 --membind=6 numactl --physcpubind=112-119,240-247 --membind=7 numactl --physcpubind=120-127,248-255 --membind=7

Table 4-1: NUMA binding (LLC as NUMA Domain disabled; 8 CCDs/processor)

BIOS (for an 12 CCD, 64- and 96-core CPU)	LLC as NUMA Domain Disabled (NPS4)
Number of NUMA nodes shown by `# numactl -H` shows	• **NUMA node(s):** 8 • **NUMA node0 CPU(s):** 0-23,192-215 • **NUMA node1 CPU(s):** 24-47,216-239 • **NUMA node2 CPU(s):** 48-71,240-263 • **NUMA node3 CPU(s):** 72-95,264-287 • **NUMA node4 CPU(s):** 96-119,288-311 • **NUMA node5 CPU(s):** 120-143,312-335 • **NUMA node6 CPU(s):** 144-167,336-359 • **NUMA node7 CPU(s):** 168-191,360-383
Number of CCDs in a NUMA node	3
8 JVMs by binding each JVM to a NUMA node	`numactl --cpunodebind=$node --membind=$node`
24 JVMs by binding JVMs to a NUMA node and using CPU cores within a NUMA node. In this example, multiple JVMs run tied to a NUMA node.	Socket 0: numactl --physcpubind=0-7,192-199 --membind=0 numactl --physcpubind=8-15,200-207 --membind=0 numactl --physcpubind=16-23,208-215 --membind=0 numactl --physcpubind=24-31,216-223 --membind=1 numactl --physcpubind=32-39,224-231 --membind=1 numactl --physcpubind=40-47,232-239 --membind=1 numactl --physcpubind=48-55,240-247 --membind=2 numactl --physcpubind=56-63,248-255 --membind=2 numactl --physcpubind=64-71,256-263 --membind=2 numactl --physcpubind=72-79,264-271 --membind=3 numactl --physcpubind=80-87,272-279 --membind=3 numactl --physcpubind=88-95,280-287 --membind=3 Socket 1: numactl --physcpubind=96-103,288-295 --membind=4 numactl --physcpubind=104-111,296-303 --membind=4 numactl --physcpubind=112-119,304-311 --membind=4 numactl --physcpubind=120-127,312-319 --membind=5 numactl --physcpubind=128-135,320-327 --membind=5 numactl --physcpubind=136-143,328-335 --membind=5 numactl --physcpubind=144-151,336-343 --membind=6 numactl --physcpubind=152-159,344-351 --membind=6 numactl --physcpubind=160-167,352-359 --membind=6 numactl --physcpubind=168-175,360-367- --membind=7 numactl --physcpubind=176-183,368-375 --membind=7 numactl --physcpubind=184-191,376-383 --membind=7

Table 4-2: NUMA binding (LLC as NUMA Domain disabled; 12 CCDs/processor)

BIOS	LLC as NUMA Domain Enabled (NPS4)
Number of NUMA nodes	16
Number of CCD per NUMA node	1
16 JVMs by binding each JVM to a NUMA node	`nodenumactl --cpunodebind=$node --localalloc`
8 JVMs by binding JVM process to two NUMA nodes	`numactl --cpunodebind=$node,$node+1 --localalloc`

Table 4-3: NUMA binding (LLC as NUMA Domain enabled; 8 CCDs/processor)

BIOS	LLC as NUMA Domain Enabled (NPS4)
Number of NUMA nodes	24
Number of CCD per NUMA node	1
24 JVMs by binding each JVM to a NUMA node	`nodenumactl --cpunodebind=$node --localalloc`
12 JVMs by binding JVM process to two NUMA nodes	`numactl --cpunodebind=$node,$node+1 --localalloc`

Table 4-4: NUMA binding (LLC as NUMA Domain enabled; 12 CCDs/processor)

4.3 - Kernel Parameters

AMD set the following Linux kernel parameters for optimal SPECjbb2015:

- **dirty_background_ratio:** Contains the number of pages upon which the background kernel flusher threads will start writing out dirty data as a percentage of total available memory that contains free pages and reclaimable pages.
- **dirty_ratio:** Contains the number of pages upon which a process which is generating disk writes will itself start writing out dirty data as a percentage of total available memory that contains free pages and reclaimable pages.
- **dirty_writeback_centisecs:** The `pdflush` writeback daemons will periodically wake up and write "old" data out to disk. This tunable expresses the interval between those wake-ups in 100th of a second. Setting this to zero disables periodic write back.
- **dirty_expire_centisecs:** Defines when dirty data is old enough to be eligible for writeout by the `pdflush` daemons, expressed in 100th of a second. Data that has been dirty in memory for longer than this interval will be written out next time when a `pdflush` daemon wakes up.

The following parameters related to kernel.sched deliver SPECjbb® 2015 performance improvements (see below for alternative parameters that do not support kernel.sched):

- **sched_migration_cost_ns:** Migration process quantum in nanoseconds before the kernel migrates it again to another core. Default is 50000.
- **sched_rt_runtime_us:** Global limit on how much time real-time scheduling may use.
- **sched_latency_ns:** Period over which CFQ tries to fairly schedule the tasks on the run queue.
- **sched_min_granularity_ns:** Minimum time a task will be allowed to run on the CPU before being preempted out. Default is 4ms.
- **sched_wakeup_granularity_ns:** Controls the wake-up latency of a task.

Linux Kernel version 6.6 and later have discontinued some kernel.sched related parameters (see https://bugzilla.redhat.com/show_bug.cgi?id=1957829*). Specifically, `sched_latency_ns`, `sched_min_granularity_ns`, and `sched_wakeup_granularity_ns` are being discontinued. The new `base_slice_ns` parameter can be set with a suitable value to model behavior that is similar to that achievable by using these three discontinued parameters. You may also consider the following alternatives:

- **base_slice_ns:** This is equivalent to `min_granularity_ns` and can be set using `/sys/kernel/debug/sched/base_slice_ns`. If you are only using `min_granularity_ns`, then setting `base_slice_ns` to same values is sufficient. However, if you are using this in combination with `wakeup_granularity_ns`, then you will need to evaluate new suitable values.
- **sched_wakeup_granularity_ns:** Earlier versions of `sched_wakeup_granularity_ns` can only be set at the system level to control wakeup latencies. However, Linux Kernel versions 6.6 or later allow users to explore settings on a per-task level to control wakeup latencies. See https://man7.org/linux/man-pages/man1/nice.1.html*. You can set this using `/sys/kernel/debug/sched/migration_cost_ns`.

The following parameter is still available through the debug path:

- **sched_migration_cost_ns:** You can set this using the path `/sys/kernel/debug/sched/migration_cost_ns`.

The following scheduler features can impact performance when using the new Earliest Eligible Virtual Deadline First (EEVDF) scheduler and legacy tunables like `wakeup_granularity_ns` are not available:

- **RUN_TO_PARITY:** Allows the task to run until its `vruntime` is equal to the average `vruntime` of the `runqueue` despite there being other eligible tasks with shorter deadline. This was done to reduce preemption because the situation of the `runqueues` change with time.
- **WAKEUP_PREEMPTION:** Disabled wakeup preemption throughout the system. Only tick driven preemption remains.

Evaluate resource limits requirements for your Java application. You must reboot the server for these settings to take effect. Resource limits were increased as follows:

- **/etc/security/limits.conf:** Set hard and soft resource limits for the following items:
  - **nofile:** Max number of open files.
  - **memlock:** Max locked-in-memory address space (KB)
  - **stack:** Max stack size (KB)

```
*       -       nofile      1024000
*       soft    memlock     unlimited
*       hard    memlock     unlimited
*       -       stack       unlimited
```

- To raise or restrict the number of processes forked, change the global value for `DefaultTasksMax=970000` in `/etc/system/system.conf`.
- To view the current effective value, execute the command `$ systemctl show --property DefaultTasksMax`
- To change the maximum number of OS tasks each user may run concurrently to this desired value, set `UserTasksMax=970000` in `/etc/system/logind.conf`.

THIS PAGE INTENTIONALLY LEFT BLANK.

Chapter 5: Resources

- Memory Population Guidelines for AMD Family 1Ah Models 00h–0Fh and Models 10h–1Fh Socket SP5 Processors - Login required; please review the latest version if multiple versions are present.
- Socket SP5/SP6 Platform NUMA Topology for AMD Family 1Ah Models 00h–0Fh and Models 10h–1Fh - Login required; please review the latest version if multiple versions are present.
- AMD I/O Virtualization Technology (IOMMU) Specification - Log in required.
- From the AMD Documentation Hub:
  - *BIOS & Workload Tuning Guide for AMD EPYC™ 9005 Series Processors*
  - *Microsoft® Windows® Server Tuning Guide for AMD EPYC™ 9005 Series Processors*
  - *Microsoft® Windows® Network Tuning Guide for AMD EPYC™ 9005 Series Processors*
  - *RedHat Enterprise Linux® Tuning Guide AMD EPYC™ 9005 Series Processors*
  - *Linux® Network Tuning Guide for AMD EPYC™ 9005 Series Processors*
  - *VMware vSphere® Network Tuning Guide for AMD EPYC™ 9005 Series Processors*
- OpenJDK*
- SPECjbb®2015*

Java® Tuning Guide for AMD EPYC™ 9005 Processors

PID: 58481

Anil Rajput, Gnanakumar Rajaram, and the AMD India Java team