- About
- Adoption
- Amber
- Build
- Client Libraries
- Code Tools
- Coin
- Compatibility & Specification Review
- Compiler
- CRaC
- Device I/O
- Duke
- Graal
- HotSpot
- IDE Tooling & Support
- JDK 8
- JDK 8u
- JDK Updates
- Kulla
- Lanai
- Lilliput
- Loom
- Memory Model Update
- Mission Control
- Multi-Language VM
- Nashorn
- OpenJFX
- Port: AArch64
- Port: BSD
- Port: MacOSX
- Port: PowerPC/AIX
- Port: RISC-V
- Port: s390x
- Portola
- Quality
- Shenandoah
- Skara
- Sumatra
- Tsan
- Type Annotations
- Valhalla
- Wakefield
- **ZGC**
  - Pointer Metadata using Multi-Mapped memory

Main

Created by Iris Clark, last modified by Stefan Karlsson on Sep 19, 2023

The Z Garbage Collector (**ZGC**) is a scalable low latency garbage collector. ZGC performs all expensive work concurrently, without stopping the execution of application threads for more than a **millisecond**. It is suitable for applications which require low latency. Pause times are **independent of the heap size** that is being used. ZGC works well with heap sizes from a few hundred megabytes to **16TB**.

ZGC was initially introduced as an experimental feature in JDK 11, and was declared **Production Ready** in JDK 15. In JDK 21 was reimplemented to support generations.

At a glance, ZGC is:

- Concurrent
- Region-based
- Compacting
- NUMA-aware
- Using colored pointers
- Using load barriers
- Using store barriers (in the generational mode)

At its core, ZGC is a **concurrent** garbage collector, meaning all heavy lifting work is done while **Java threads continue to execute**. This greatly limits the impact garbage collection will have on your application's response time.

This OpenJDK project is sponsored by the HotSpot Group.

Download
Latest Stable: JDK 21
Latest Development: JDK 22 Early Access

Source Code
github.com/openjdk/jdk

Blog Posts
ZGC | What's new in JDK 18
ZGC | What's new in JDK 17
ZGC | What's new in JDK 16
ZGC | What's new in JDK 15
ZGC | Using -XX:SoftMaxHeapSize
ZGC | What's new in JDK 14
Compact Forwarding Information
How do 'hot and cold' objects behave?

Talks/Presentations/Podcasts
JVMLS 2023 - Video (33 min)

Oracle Developer Live 2022 - Slides | Video (28 min)
Jokerconf 2021 - Slides
Inside Java Podcast - ZGC - Sound (30 min)
Oracle Developer Live 2020 - Slides | Video (40 min)
Oracle Code One 2019 - Slides
PL-Meetup 2019 - Slides
Jfokus 2019 - Slides | Video (21 min)
Devoxx 2018 - Slides | Video (40 min)
Oracle Code One 2018 - Slides | Video (45 min)
Jfokus 2018 - Slides | Video (45 min)
FOSDEM 2018 - Slides

Mailing List
Subscribe | Archive

Project
Members
JIRA Dashboard
JEPs 333, 351, 364, 365, 376, 377, 439

Contents

Supported Platforms

Platform	Supported	Since	Comment
Linux/x64	✔	JDK 15 (Experimental since JDK 11)	
Linux/AArch64	✔	JDK 15 (Experimental since JDK 13)	
Linux/PowerPC	✔	JDK 18	
macOS/x64	✔	JDK 15 (Experimental since JDK 14)	
macOS/AArch64	✔	JDK 17	
Windows/x64	✔	JDK 15 (Experimental since JDK 14)	Requires Windows version 1803 (Windows 10 or Windows Server 2019) or later.
Windows/AArch64	✔	JDK 16	

Quick Start

If you're trying out ZGC for the first time, start by using the following GC options:

```
-XX:+UseZGC -XX:+ZGenerational -Xmx<size> -Xlog:gc
```

For more detailed logging, use the following options:

```
-XX:+UseZGC -XX:+ZGenerational -Xmx<size> -Xlog:gc*
```

See below for more information on these and additional options.

Configuration & Tuning

ZGC has been designed to be adaptive and to require minimal manual configuration. During the execution of the Java program, ZGC dynamically adapts to the workload by resizing generations, scaling the number of GC threads, and adjusting tenuring thresholds. The main tuning knob is to increase the maximum heap size.

ZGC comes in two versions: The new, generational version and the legacy, non-generational version. The Non-generational ZGC is the older version of ZGC, which doesn't take advantage of generations (see Generations) to optimize its runtime characteristics. It is encouraged that users transition to use the newer Generational ZGC.

The Generational ZGC is enabled with the command-line options `-XX:+UseZGC -XX:+ZGenerational`.

The Non-generational ZGC is enabled with the command-line option `-XX:+UseZGC`.

Overview

The following JVM options can be used with ZGC:

-XX:MinHeapSize, -Xms	-XX:ZAllocationSpikeTolerance	-XX:ZStatisticsInterval
-XX:InitialHeapSize, -Xms	-XX:ZCollectionInterval	-XX:ZVerifyForwarding
-XX:MaxHeapSize, -Xmx	-XX:ZFragmentationLimit	-XX:ZVerifyMarking
-XX:SoftMaxHeapSize	-XX:ZMarkStackSpaceLimit	-XX:ZVerifyObjects
-XX:ConcGCThreads	-XX:ZProactive	-XX:ZVerifyRoots
-XX:ParallelGCThreads	-XX:ZUncommit	-XX:ZVerifyViews
-XX:UseDynamicNumberOfGCThreads	-XX:ZUncommitDelay	-XX:ZYoungGCThreads
-XX:UseLargePages		-XX:ZOldGCThreads
-XX:UseTransparentHugePages		-XX:ZBufferStoreBarriers
-XX:UseNUMA		
-XX:SoftRefLRUPolicyMSPerMB		
-XX:AllocateHeapAt		

In addition to these the following flags are available when the generational mode is enabled with -XX:+UseZGC -XX:+ZGenerational:

General GC Options	ZGC Options	ZGC Diagnostic Options (-XX:+UnlockDiagnosticVMOptions)
	-XX:ZCollectionIntervalMinor	-XX:ZVerifyRemembered
	-XX:ZCollectionIntervalMajor	-XX:ZYoungGCThreads
	-XX:YoungCompactionLimit	-XX:ZOldGCThreads
		-XX:ZBufferStoreBarriers

Setting Heap Size

The most important tuning option for ZGC is setting the maximum heap size, which you can set with the `-Xmx` command-line option. Because ZGC is a concurrent collector, y maximum heap size such that the heap can accommodate the live-set of your application and there is enough headroom in the heap to allow allocations to be serviced while much headroom is needed very much depends on the allocation rate and the live-set size of the application. In general, the more memory you give to ZGC the better. But at memory is undesirable, so it's all about finding a balance between memory usage and how often the GC needs to run.

ZGC has another command-line option related to the heap size named `-XX:SoftMaxHeapSize`. It can be used to set a soft limit on how large the Java heap can grow. ZGC w beyond this limit, but is still allowed to grow beyond this limit up to the maximum heap size. ZGC will only use more than the soft limit if that is needed to prevent the Java ap waiting for the GC to reclaim memory. For example, with the command-line options `-Xmx5g -XX:SoftMaxHeapSize=4g` ZGC will use 4GB as the limit for its heuristics, but if it c below 4GB it is still allowed to temporarily use up to 5GB.

Setting Concurrent GC Threads

Note! This section pertain to the non-generational version of ZGC. Generational ZGC has a more adaptive implementation and you are less likely to need to tweak the GC t

The second tuning option one might want to look at is setting the number of concurrent GC threads (`-XX:ConcGCThreads=<number>`). ZGC has heuristics to automatically sele heuristic usually works well but depending on the characteristics of the application this might need to be adjusted. This option essentially dictates how much CPU-time the G too much and the GC will steal too much CPU-time from the application. Give it too little, and the application might allocate garbage faster than the GC can collect it.

NOTE!! Starting from JDK 17, ZGC dynamically scales up and down the number of concurrent GC threads. This makes it even more unlikely that you'd need to adjust the co threads.

NOTE!!! In general, if low latency (i.e. low application response time) is important for you application, then *never* over-provision your system. Ideally, your system should nev CPU utilization.

Returning Unused Memory to the Operating System

By default, ZGC uncommits unused memory, returning it to the operating system. This is useful for applications and environments where memory footprint is a concern, but i impact on the latency of Java threads. You can disable this feature with the command-line option `-XX:-ZUncommit`. Furthermore, memory will not be uncommitted so that the the minimum heap size (`-Xms`). This means this feature will be implicitly disabled if the minimum heap size (`-Xms`) is configured to be equal to the maximum heap size (`-Xmx`

You can configure an uncommit delay using `-XX:ZUncommitDelay=<seconds>` (default is 300 seconds). This delay specifies for how long memory should have been unused b uncommit.

NOTE! Allowing the GC to commit and uncommit memory while the application is running could have a negative impact on the latency of Java threads. If extremely low later running with ZGC, consider running with the same value for `-Xmx` and `-Xms`, and use `-XX:+AlwaysPreTouch` to page in memory before the application starts.

NOTE!! On Linux, uncommitting unused memory requires `fallocate(2)` with `FALLOC_FL_PUNCH_HOLE` support, which first appeared in kernel version **3.5** (for tmpfs) and **4.3** (f

Using Large Pages

Configuring ZGC to use large pages will generally yield better performance (in terms of throughput, latency and start up time) and comes with no real disadvantage, except t complicated to setup. The setup process typically requires root privileges, which is why it's not enabled by default.

Enabling Large Pages On Linux

On Linux x86, large pages (also known as "huge pages") have a size of 2MB.

Let's assume you want a 16GB Java heap. That means you need 16GB / 2MB = 8192 huge pages.

The heap requires at least 16GB (8192 pages) of memory to the pool of huge pages. The heap along with other parts of the JVM will use large pages for various internal dat code heap and marking bitmaps). In this example you will reserve 9216 pages (18GB) to allow for 2GB of non-Java heap allocations to use large pages.

Configure the system's huge page pool to have the required number of pages (requires root privileges):

```
$ echo 9216 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

Note that the above command is not guaranteed to be successful if the kernel cannot find enough free huge pages to satisfy the request. Also note that it might take some ti process the request. Before proceeding, check the number of huge pages assigned to the pool to make sure the request was successful and has completed.

```
$ cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
9216
```

NOTE! If you're using a **Linux kernel >= 4.14**, then the next step (where you mount a hugetlbfs filesystem) can be skipped. However, if you're using an older kernel then ZG pages through a hugetlbfs filesystem.

Mount a hugetlbfs filesystem (requires root privileges) and make it accessible to the user running the JVM (in this example we're assuming this user has 123 as its uid).

```
$ mkdir /hugepages
$ mount -t hugetlbfs -o uid=123 nodev /hugepages
```

Now start the JVM using the `-XX:+UseLargePages` option.

```
$ java -XX:+UseZGC -Xms16G -Xmx16G -XX:+UseLargePages ...
```

If there are more than one accessible hugetlbfs filesystem available, then (and only then) do you also have to use `-XX:AllocateHeapAt` to specify the path to the filesystems example, assume there are multiple accessible hugetlbfs filesystems mounted, but the filesystem you specifically want to use it mounted on `/hugepages`, then use the followi

NOTE! The configuration of the huge page pool and the mounting of the hugetlbfs file system is not persistent across reboots, unless adequate measures are taken.

Enabling Transparent Huge Pages On Linux

NOTE! On Linux, using ZGC with transparent huge pages enabled requires **kernel >= 4.7**.'

Use the following options to enable transparent huge pages in the VM:

```
-XX:+UseLargePages -XX:+UseTransparentHugePages
```

These options tell the JVM to issue `madvise(..., MADV_HUGEPAGE)` calls for memory it maps, which is useful when using transparent huge pages in `madvise` mode.

To enable transparent huge pages, you also need to configure the kernel by enabling `madvise` mode.

```
$ echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
```

ZGC uses `shmem` huge pages for the heap, so the following kernel setting also needs to be configured:

```
$ echo advise > /sys/kernel/mm/transparent_hugepage/shmem_enabled
```

It is important to check these kernel settings when comparing the performance of different GCs. Some Linux distributions forcefully enable transparent huge pages for privat
`/sys/kernel/mm/transparent_hugepage/enabled` to be set to `always`, while leaving `/sys/kernel/mm/transparent_hugepage/shmem_enabled` at the default `never`. In this case a
use of transparent huge pages for the heap. See Transparent Hugepage Support for more information.

Enabling NUMA Support

ZGC has NUMA support, which means it will try it's best to direct Java heap allocations to NUMA-local memory. This feature is **enabled by default**. However, it will automati
JVM detects that it's bound to only use memory on a single NUMA node. In general, you don't need to worry about this setting, but if you want to explicitly override the JVM's
by using the `-XX:+UseNUMA` or `-XX:-UseNUMA` options.

When running on a NUMA machine (e.g. a multi-socket x86 machine), having NUMA support enabled will often give a noticeable performance boost.

Enabling GC Logging

GC logging is enabled using the following command-line option:

```
-Xlog:<tag set>,[<tag set>, ...]:<log file>
```

For general information/help on this option:

```
-Xlog:help
```

To enable basic logging (one line of output per GC):

```
-Xlog:gc:gc.log
```

To enable GC logging that is useful for tuning/performance analysis:

```
-Xlog:gc*:gc.log
```

Where `gc*` means log all tag combinations that contain the `gc` tag, and `:gc.log` means write the log to a file named `gc.log`.

Change Log

JDK 21

- Support for generations (-XX:+ZGenerational) (JEP 439)

JDK 18

- Support for String Deduplication (-XX:+UseStringDeduplication)
- Linux/PowerPC support
- Various bug-fixes and optimizations

JDK 17

- Dynamic Number of GC threads
- Reduced mark stack memory usage
- macOS/aarch64 support
- GarbageCollectorMXBeans for both pauses and cycles
- Fast JVM termination

JDK 16

- Concurrent Thread Stack Scanning (JEP 376)
- Support for in-place relocation
- Performance improvements (allocation/initialization of forwarding tables, etc)

JDK 15

- Production ready (JEP 377)
- Improved NUMA awareness
- Improved allocation concurrency
- Support for Class Data Sharing (CDS)
- Support for placing the heap on NVRAM
- Support for compressed class pointers
- Support for incremental uncommit
- Fixed support for transparent huge pages
- Additional JFR events

JDK 14

- macOS support (JEP 364)
- Windows support (JEP 365)
- Support for tiny/small heaps (down to 8M)
- Support for JFR leak profiler
- Support for limited and discontiguous address space
- Parallel pre-touch (when using -XX:+AlwaysPreTouch)
- Performance improvements (clone intrinsic, etc)
- Stability improvements

- Support for uncommitting unused memory (JEP 351)
- Support for -XX:SoftMaxHeapSIze
- Support for the Linux/AArch64 platform
- Reduced Time-To-Safepoint

JDK 12

- Support for concurrent class unloading
- Further pause time reductions

JDK 11

- Initial version of ZGC
- Does not support class unloading (using -XX:+ClassUnloading has no effect)

FAQ

What does the "Z" in ZGC stand for?

It doesn't stand for anything, ZGC is just a name. It was originally inspired by, or a homage to, ZFS (the filesystem) which in many ways was revolutionary when it first came an acronym for "Zettabyte File System", but that meaning was abandoned and it was later said to not stand for anything. It's just a name. See Jeff Bonwick's Blog for more d

Is it pronounced "zed gee see" or "zee gee see"?

© 2024 Oracle Corporation and/or its affiliates

Terms of Use • License: GPLv2 • Privacy • Trademarks • Contact Us

Powered by a free **Atlassian Confluence Open Source Project License** granted to https://www.atlassian.com/software/views/opensource-community-additional-license-offer. Evaluate Confluence today.