JEP 439: Generational ZGC

Owner	Stefan Karlsson
Type	Feature
Scope	Implementation
Status	Completed
Release	21
Component	hotspot / gc
Discussion	hotspot dash gc dash dev at openjdk dot org
Effort	XL
Duration	XL
Relates to	JEP 377: ZGC: A Scalable Low-Latency Garbage Collector (Production)
Reviewed by	Erik Helin, Erik Österlund, Vladimir Kozlov
Endorsed by	Vladimir Kozlov
Created	2021/08/25 12:01
Updated	2023/08/14 20:24
Issue	8272979

Summary

Improve application performance by extending the Z Garbage Collector (ZGC) to maintain separate generations for young and old objects. This will allow ZGC to collect young objects — which tend to die young — more frequently.

Goals

Applications running with Generational ZGC should enjoy

- Lower risks of allocations stalls,
- Lower required heap memory overhead, and
- Lower garbage collection CPU overhead.

These benefits should come without significant throughput reduction compared to non-generational ZGC. The essential properties of non-generational ZGC should be preserved:

- Pause times should not exceed 1 millisecond,
- Heap sizes from a few hundred megabytes up to many terabytes should be supported, and
- Minimal manual configuration should be needed.

As examples of the last point, there should be no need to manually configure

- The size of the generations,
- The number of threads used by the garbage collector, or
- For how long objects should reside in the young generation.

Finally, Generational ZGC should be a better solution for most use cases than non-generational ZGC. We should eventually be able to replace the latter with the former in order to reduce long-term maintenance costs.

Non-Goals

It is not a goal to perform reference processing in the young generation.

Motivation

ZGC (JEP 333) is designed for low latency and high scalability. It has been available for production use since JDK 15 (JEP 377).

ZGC does the majority of its work while application threads are running, pausing those threads only briefly. ZGC's pause times are consistently measured in microseconds; by contrast the pause times of the default garbage collector, G1, range from milliseconds to seconds. ZGC's low pause times are independent of heap size: Workloads can use heap sizes from a few hundred megabytes all the way up to multiple terabytes and still enjoy low pause times.

For many workloads, simply using ZGC is enough to solve all latency problems related to garbage collection. This works well as long as there are sufficient resources (i.e., memory and CPU) available to ensure that ZGC can reclaim memory faster than the concurrently-running application threads consume it. However, ZGC currently stores all objects together, regardless of age, so it must collect all objects every time it runs.

The weak generational hypothesis states that young objects tend to die young, while old objects tend to stick around. Thus collecting young objects requires fewer resources and yields more memory, while collecting old objects requires more resources and yields less memory. We can thus improve the performance of applications that use ZGC by collecting young objects more frequently.

Description

Enabling Generational ZGC

To ensure a smooth succession, we will initially make Generational ZGC available alongside non-generational ZGC. The `-XX:+UseZGC` command-line option will select non-generational ZGC; to select Generational ZGC, add the `-XX:+ZGenerational` option:

```
$ java -XX:+UseZGC -XX:+ZGenerational ...
```

In a future release we intend to make Generational ZGC the default, at which point `-XX:-ZGenerational` will select non-generational ZGC. In an even later release we intend to remove non-generational ZGC, at which point the `ZGenerational` option will become obsolete.

Design

Generational ZGC splits the heap into two logical *generations*: The *young* generation is for recently allocated objects while the *old* generation is for long-lived objects. Each generation is collected independently of the other, so ZGC can focus on collecting the profitable young objects.

As in non-generational ZGC, all garbage collection is done concurrently while the application is running, with application pauses typically shorter than one millisecond. Since ZGC reads and modifies the object graph at the same time as the application, it must take care to give the application a consistent view of the object graph. ZGC does this via colored pointers, load barriers, and store barriers.

- A *colored pointer* is a pointer to an object in the heap which, along with the object's memory address, includes metadata that encodes the known state of the object. The metadata describes whether the object is known to be alive, whether the address is correct, and so forth. ZGC always uses 64-bit object pointers and can therefore accommodate metadata bits and object addresses for heaps up to many terabytes. When a field in an object refers to another object, ZGC implements that reference with a colored pointer.
- A *load barrier* is a fragment of code injected by ZGC into the application wherever the application reads a field of an object that refers to another object. The load barrier interprets the metadata of the colored pointer stored in the field and potentially takes some action before the application uses the referenced object.

Non-generational ZGC uses both colored pointers and load barriers. Generational ZGC also uses store barriers to efficiently keep track of references from objects in one generation to objects in another generation.

- A *store barrier* is a fragment of code injected by ZGC into the application wherever the application stores references into object fields. Generational ZGC adds new metadata bits to colored pointers so that store barriers can determine if the field being written has already been recorded as potentially containing an inter-generational pointer. Colored pointers make Generational ZGC's store barriers more efficient than traditional generational store barriers.

The addition of store barriers allows Generational ZGC to move the work of marking reachable objects from load barriers to store barriers. That is, store barriers can use the metadata bits in colored pointers to efficiently determine if the object previously referred to by the field, prior to the store, needs to be marked.

Moving marking out of load barriers makes it easier to optimize them, which is important because load barriers are often more frequently executed than store barriers. Now when a load barrier interprets a colored pointer it need only update the object address, if the object was relocated, and update the metadata to indicate that the address is known to be correct. Subsequent load barriers will interpret this metadata and not check again whether the object has been relocated.

Generational ZGC uses distinct sets of marking and relocation metadata bits in colored pointers, so that the generations can be collected independently.

The remaining sections describe important design concepts that distinguish Generational ZGC from non-generational ZGC, and from other garbage collectors:

- No multi-mapped memory
- Optimized barriers
- Double-buffered remembered sets
- Relocations without additional heap memory
- Dense heap regions
- Large objects
- Full garbage collections

No multi-mapped memory

Non-generational ZGC uses multi-mapped memory to reduce the overhead of load barriers. Generational ZGC instead uses explicit code in the load and store barriers.

For users, the main advantage of this change is that makes it easier to measure the amount of memory used for the heap. With multi-mapped memory the same heap memory is mapped into three separate virtual address ranges, so the heap usage reported by tools such as `ps` is around triple the amount of memory actually used.

For the GC itself, this change means that the metadata bits in a colored pointer no longer need to be in the part of the pointer that corresponds to the range of accessible memory addresses for the heap. This allows more metadata bits to be added and also opens the possibility of increasing the maximum heap size beyond the 16 terabyte limit of non-generational ZGC.

In Generational ZGC, object references stored in object fields are implemented as colored pointers. Object references stored in the JVM stack, however, are implemented as *colorless* pointers, without metadata bits, in the hardware stack or

in CPU registers. The load and store barriers translate back and forth between colored and colorless pointers.

Since colored pointers are never present on the hardware stack or in CPU registers, a more exotic colored pointer layout can be used as long as the conversion between colored and colorless pointers can be done efficiently. The colored pointer layout used by Generational ZGC puts the metadata in the low-order bits of the pointer and the object address in the high-order bits. This minimizes the number of machine instructions in the load barrier. With careful encoding of the memory address and metadata bits, a single shift instruction (on x64) can both check whether the pointer requires processing and remove the metadata bits.

Optimized barriers

With the introduction of store barriers, and new responsibilities for load barriers, more GC code will be inter-mixed with compiled application code. The barriers need to be highly optimized in order to maximize throughput. Many of the key design decisions of Generational ZGC involve the colored pointer scheme and the barriers.

Some of the techniques used to optimize the barriers are:

- Fast paths and slow paths
- Minimizing load barrier responsibilities
- Remembered-set barriers
- SATB marking barriers
- Fused store barrier checks
- Store barrier buffers
- Barrier patching

Fast paths and slow paths

ZGC splits barriers into two parts. The *fast path* checks whether some additional GC work must be done before the referenced object is used by the application. The *slow path* does that additional work. All object accesses run the fast-path check. As the name indicates, it needs to be fast, so this code is inserted directly into just-in-time-compiled application code. The slow path is taken only a fraction of the time. When the slow path is taken, the color of the accessed object pointer is changed so that subsequent accesses to the same pointer do not trigger the slow path again for a while. Because of this, it is less important that the slow paths be highly optimized. For maintainability, they are implemented as C++ functions in the JVM.

In non-generational ZGC, this is how load barriers are split. In Generational ZGC, the same scheme is applied to store barriers and their associated GC work.

Minimizing load barrier responsibilities

In non-generational ZGC the load barrier is responsible for

- Updating stale pointers to objects that the GC relocated and
- Marking loaded objects as being alive — the application is loading the object, so it is considered alive.

In Generational ZGC we must keep track of two generations and translate between colored and colorless pointers. To minimize complexity, and to allow optimization of the load barrier fast path, responsibility for marking is shifted to the store barriers.

In Generational ZGC the load barriers are responsible for

- Removing metadata bits from colored pointers and
- Updating stale pointers to objects that the GC relocated.

The store barriers are responsible for

- Adding metadata bits to create colored pointers,
- Maintaining the *remembered set*, which tracks old-to-young generation object pointers, and
- Marking objects as being alive.

Remembered-set barriers

When Generational ZGC collects the young generation, it only visits objects in the young generation. However, objects in the old generation can contain fields pointing to objects in the young generation, i.e., old-to-young generation pointers. These fields must be visited during a young-generation collection for two reasons.

- *GC marking roots* — Such a field can contain the only reference keeping part of the young-generation object graph reachable. The GC must treat these fields as roots of the object graph in order to ensure that all live objects are found and marked alive.
- *Stale pointers in the old generation* — Collecting the young generation moves objects, but pointers to those objects are not eagerly updated. Instead the pointers are lazily updated, by load barriers, when the application encounters them. At some point the GC must update any stale old-to-young generation pointers that the application did not encounter.

The set of old-to-young generation pointers is called the *remembered set*. The remembered set contains all memory addresses in the old generation that potentially contain pointers to objects in the young generation. Store barriers add entries to the remembered set. Whenever a reference is stored into an object field, it is considered to potentially contain an old-to-young generation pointer. The store barrier slow path filters out stores to fields in the young generation, since only addresses in the old generation are of interest. The slow path does not filter based on the value written to the field, which could be a value referring to either the

young or old generation. The GC checks the current value of the object field when it uses the remembered set.

All of this ensures the *act-once property* of store barriers with respect to maintaining the remembered set. This means that, between the start of two consecutive young-generation marking phases, the store barrier slow path is taken just once per stored-to object field. The first time a field is written to, the following steps occur:

- The fast path checks the to-be-overwritten value of the stored-to field,
- The color shows that this field has not been written to since the previous young generation marking phase, so
- The slow path is taken,
- The address of the stored-to field is added to the remembered set, and
- The new pointer value is colored and stored in the field.

The new pointer value is colored in such a way that subsequent fast-path checks will see that this object field has already passed the slow path.

SATB marking barriers

Unlike non-generational ZGC, Generational ZGC uses a *snap shot at the beginning* (SATB) marking algorithm. At the beginning of the marking phase the GC takes a snapshot of the GC roots; by the end of the marking phase, all objects reachable from those roots when marking began are guaranteed to be found and marked alive.

To accomplish this the GC needs to be notified when references between objects in the object graph are broken. Store barriers therefore report to-be-overwritten field values to the GC; the GC then marks the referenced object and visits and marks the objects reachable from it.

Store barriers need only report a to-be-overwritten field value the first time that the field is stored to within a marking cycle. Subsequent stores to the same field will only replace values that the GC is guaranteed to find anyway, because of the SATB property. The SATB property, in turn, supports the act-once property of store barriers with respect to marking.

Fused store barrier checks

There are many similarities between the remembered-set maintenance and marking functions of store barriers. Both use colored-pointer fast-path checks and their respective act-once properties. Instead of having separate fast-path checks for each condition, we fuse them into one combined fast-path check. If either of the two properties fails, the slow path is taken and the required GC work is done.

Store barrier buffers

Splitting barriers into fast paths and slow paths, and using pointer coloring, reduces the number of calls to the C++ slow-path functions. Generational ZGC reduces overhead further by placing a JIT-compiled *medium path* between the fast path and the slow path. The medium path stores the to-be-overwritten value and the address of the object field in a *store barrier buffer* and returns to the compiled application code without taking the expensive slow path. The slow path is taken only when the store barrier buffer is full. This amortizes some of the overhead of transitioning from compiled application code to the C++ slow-path code.

Barrier patching

Both load barriers and store barriers perform their checks against values in global or thread-local variables which the GC changes when it transitions to a new phase. There are different ways to read these variables in barriers, and the overhead of doing so is different on different CPU architectures.

In Generational ZGC we reduce this overhead by patching barrier code when possible. The global values are encoded in the machine instructions of barriers as immediate values. This removes the need to dereference a global or thread-local variable in order to fetch the current value. The immediate values are patched when methods are invoked the first time after the GC changes phase; e.g., when the GC starts the young generation marking phase. This further reduces barrier overhead.

Double-buffered remembered sets

Many GCs use a remembered-set technique called *card table marking* to keep track of inter-generational pointers. When an application thread writes to an object field it also writes (i.e., *dirties*) a byte in a large byte array called the *card table*. Typically, one byte in the table corresponds to an address range spanning 512 bytes in the heap. To find all the old-to-young generation object pointers, the GC must locate and visit all object fields within the address ranges that correspond to the dirty bytes in the card table.

Generational ZGC, by contrast, records object field locations precisely by using bitmaps in which each bit represents a single potential object field address. Each old-generation region has a pair of remembered-set bitmaps. One of the bitmaps is active and populated by application threads running their store barriers, while the other bitmap is used by the GC as a read-only copy of all recorded old generation object fields that potentially point to objects in the young generation. These two bitmaps are atomically swapped each time a young generation collection starts. One benefit of this approach is that application threads don't have to wait for the bitmaps to be cleared. The GC processes and then clears one of the bitmaps while the other bitmap is concurrently being populated by application threads. Another benefit is that, since this allows application threads and GC threads to work on distinct bitmaps, it removes the need for extra memory barriers between the two

types of threads. Other generational collectors that use card table marking, such as G1, require a memory fence when cards are marked, resulting in potentially worse store barrier performance.

Relocations without additional heap memory

Young-generation collections in other HotSpot GCs use a *scavenging* model where live objects are found and relocated in a single pass. All objects in the young generation must be relocated before the GC has complete knowledge about what objects were alive. GCs using this model can reclaim memory only after all objects have been relocated. Thus these GCs need to guess the amount of memory needed for the surviving objects and ensure that said amount of memory is available when the GC starts. If the guess is wrong then a more expensive cleanup operation is needed; e.g., in-place pinning of non-relocated objects, which leads to fragmentation, or a full GC with all application threads stopped.

Generational ZGC uses two passes: The first visits and marks all reachable objects, and the second relocates marked objects. Because the GC has complete liveness information before the relocation phase starts, it can partition the relocation work on a per-region granularity. As soon as all live objects have been relocated out of a region, i.e., the region has been *evacuated*, that region can be reused as a new target region for relocations or for allocations by application threads. Even when there are no more free regions to relocate objects into, ZGC can still proceed by compacting objects into the currently relocated regions. This allows Generational ZGC to relocate and compact the young generation without using additional heap memory.

Dense heap regions

When relocating objects out of the young generation, the number of live objects and the amount of memory they occupy will differ across regions. For example, more-recently allocated regions will likely contain more live objects.

ZGC analyzes the density of young-generation regions in order to determine which regions are worth evacuating and which regions are either too full or too expensive to evacuate. The regions that are not selected for evacuation are aged in place: Their objects remain at their locations and the regions are either kept in the young generation as survivor regions or promoted into the old generation. The objects in the surviving regions get a second chance to die in the hope that, by the time the next young-generation collection starts, enough objects will have died to make more of these regions eligible for evacuation.

This method of aging dense regions in place decreases the effort required to collect the young generation.

Large objects

ZGC already handles large objects well. By decoupling virtual memory from physical memory and over-reserving virtual memory, ZGC can usually dodge the fragmentation problems that sometimes make it difficult to allocate large objects when using G1.

In Generational ZGC we take this a step further by allowing large objects to be allocated in the young generation. Given that regions can be aged without relocating them, there is no need to allocate large objects in the old generation just to prevent expensive relocations. Instead, they can be collected in the young generation if they are short-lived or be cheaply promoted to the old generation if they are long-lived.

Full garbage collections

When the old generation is collected, there will be pointers from objects in the young generation to objects in the old generation. These pointers are considered roots of the old-generation object graph. Objects in the young generation mutate often, so young-to-old generation pointers are not tracked. Instead these pointers are found by running a young-generation collection along with the old-generation marking phase. When this young-generation collection finds pointers into the old generation it passes them to the old-generation marking process.

This extra young-generation collection will still execute as a normal young-generation collection and leave live objects in the surviving regions. One effect of this is that surviving objects in the young generation will not be subject to the reference processing and class unloading done when collecting the old generation. This could be observed by an application that, for example, releases the last reference to an object graph, invokes `System.gc()`, and then expects some weak reference to be cleared or enqueued or some class to be unloaded. To mitigate this, when a GC is requested explicitly by application code then an extra young-generation collection is done first, before the old-generation collection starts, to promote all surviving objects into the old generation.

Alternatives

Simpler barrier and pointer-coloring schemes

The current load and store barrier implementation is non-trivial to understand. A simpler version could be easier to maintain at the cost of more expensive load and store barriers. We evaluated around ten different barrier implementations; none was as performant as the chosen shift-based load barrier. Continued investigation and analysis of this performance vs. complexity trade-off might still be worth considering.

Keep using multi-mapped memory

The colorless-roots scheme could be skipped by using a simpler solution that utilizes multi-mapped memory. If more metadata bits in the pointers are needed than in non-generational ZGC, then the maximum heap size would be restricted. Another approach could be to use a hybrid solution, with some bits using multi-mapped memory and other bits being removed and added by the load and store barriers.

Testing

The ZGC implementation uses distinct C++ types for colorless and colored pointers which ensure that no implicit conversion can be made between the two types. Colored pointers are restricted to the GC code and to barriers. As long as the runtime system uses HotSpot's access API and barriers to access object pointers, it will only ever see dereferenceable colorless pointers. The runtime-visible object pointer type will always contain colorless pointers. We inject extensive verification code into the different object-pointer types to quickly find when pointers are broken or barriers are missing.

- The standard set of tests for garbage collection algorithms will be used to demonstrate correctness.

Risks and Assumptions

Implementation complexity

The barriers and colored pointers used in Generational ZGC are more complex than those in non-generational ZGC. Generational ZGC also runs two garbage collectors concurrently; these collectors are fairly independent but they do interact in a few intricate ways, adding to the complexity of the implementation.

Given the extra complexity, in the long term we intend to minimize maintenance costs by fully replacing the original, non-generational version of ZGC with Generational ZGC.

Generational ZGC will perform differently than non-generational ZGC

We believe that Generational ZGC will be better suited than its predecessor to most use cases. Some workloads might even experience a throughput improvement with Generational ZGC due to lower resource usage. For example, when running an Apache Cassandra benchmark Generational ZGC requires a quarter of the heap size yet achieves four times the throughput compared to non-generational ZGC, while still keeping pause times under one millisecond.

Some workloads are non-generational by nature and could see a slight performance degradation. We believe that this is a sufficiently small set of workloads that it does not justify the cost of maintaining two separate versions of ZGC over the long term.

One additional source of overhead is the more capable GC barriers. We expect that most of this will be offset by the gains of not having to frequently collect objects in the old generation.

Another additional source of overhead is having two garbage collectors running concurrently. We need to make sure to balance their invocation rates and CPU consumption so that they do not unduly affect the application.

As is normal for GC development, future improvements and optimizations will be driven by benchmarks and user feedback. We intend to keep improving Generational ZGC even after the first release.