DOMINIK INFÜHR

A FIRST LOOK INTO ZGC

Jan 3, 2018 • 13 minutes read

ZGC is a new garbage collector recently open-sourced by Oracle for the OpenJDK. It was mainly written by Per Liden. ZGC is similar to Shenandoah or Azul's C4 that focus on reducing pause-times while still compacting the heap. Although I won't give a full introduction here, "compacting the heap" just means moving the still-alive objects to the start (or some other region) of the heap. This helps to reduce fragmentation but usually this also means that the whole application (that includes all of its threads) needs to be halted while the GC does its magic, this is usually referred to as *stopping the world*. Only when the GC is finished, the application can be resumed. In GC literature the application is often called *mutator*, since from the GC's point of view the application mutates the heap. Depending on the size of the heap such a pause could take several seconds, which could be quite problematic for interactive applications.

There are several ways to reduce pause times:

- The GC can employ multiple threads while compacting (*parallel* compaction).
- Compaction work can also be split across multiple pauses (*incremental* compaction).
- Compact the heap concurrently to the running application without stopping it (or just for a short time) (*concurrent* compaction).
- Go's GC simply deals with it by not compacting the heap at all.

As already mentioned ZGC does concurrent compaction, this is certainly not a simple feature to implement so I want to describe how this works. Why is this complicated?

- You need to copy an object to another memory address, at the same time another thread could read from or write into the old object.
- If copying succeeded there might still be arbitrary many references somewhere in the heap to the old object address that need to be updated to the new address.

I should also mention that although concurrent compaction seems to be the best solution to reduce pause time of the alternatives given above, there are definitely some tradeoffs involved. So if you don't care about pause times, you might be better off using a GC that focuses on throughput instead.

GC barriers

The key to understanding how ZGC does concurrent compaction is the *load barrier* (often called *read barrier* in GC literature). Although I have an own section about ZGC's load-barrier, I want to give a short overview since not all readers might be familiar with them. If a GC has load-barriers, the GC needs to do some additional action when reading a reference from the heap. Basically in Java every time you see some code like `obj.field`. A GC could also need a *write/store-barrier* for operations like `obj.field = value`. Both operations are special since they read from or write into the heap. The names are a bit confusing, but GC barriers are different from memory barriers for the CPU.

Both reading and writing in the heap is extremely common, so both GC-barriers need to be super efficient. That means just a few assembly instructions in the common case. Read barriers are an order of magnitude more likely than write-barriers (although this can certainly vary depending on the application), so read-barriers are even more

performance-sensitive. Generational GC's for example usually get by with just a write barrier, no read barrier needed. ZGC needs a read barrier but no write barrier. For concurrent compaction I haven't seen a solution without read barriers.

Another factor to consider: Even if a GC needs some type of barrier, they might "only" be required when reading or writing references in the heap. Reading or writing primitives like `int` or `double` might not require the barrier.

Pointer tagging

ZGC stores additional metadata in heap references, on x64 a reference is 64-bit wide (ZGC doesn't support compressed oops or class pointers at the moment). 48-bit of those 64-bit can actually be used for virtual memory addresses on x64. Although to be exact only 47-bit, since bit 47 determines the value of bits 48-63 (for our purpose those bits are all 0). ZGC reserves the first 42-bits for the actual address of the object (referenced to as *offset* in the source code). 42-bit addresses give you a theoretical heap limitation of 4TB in ZGC. The remaining bits are used for these flags: `finalizable`, `remapped`, `marked1` and `marked0` (one bit is reserved for future use). There is a really nice ASCII drawing in ZGC's source that shows all these bits:

```
 6                 4 4 4  4 4                                             0
 3                 7 6 5  2 1                                             0
+-------------------+-+----+-----------------------------------------------+
|00000000 00000000 0|0|1111|11 11111111 11111111 11111111 11111111 11111111|
+-------------------+-+----+-----------------------------------------------+
|                   | |    |
|                   | |    * 41-0 Object Offset (42-bits, 4TB address space)
|                   | |
|                   | * 45-42 Metadata Bits (4-bits)  0001 = Marked0
```

```
|                  |                                  0010 = Marked1
|                  |                                  0100 = Remapped
|                  |                                  1000 = Finalizable
|                  |
|                  * 46-46 Unused (1-bit, always zero)
|
* 63-47 Fixed (17-bits, always zero)
```

Having metadata information in heap references does make dereferencing more expensive, since the address needs to be masked to get the *real* address (without metainformation). ZGC employs a nice trick to avoid this: When reading from memory exactly one bit of `marked0`, `marked1` or `remapped` is set. When allocating a page at offset `x`, ZGC maps the same page to 3 different address:

1. for `marked0`: `(0b0001 << 42) | x`
2. for `marked1`: `(0b0010 << 42) | x`
3. for `remapped`: `(0b0100 << 42) | x`

ZGC therefore just reserves 16TB of address space (but not actually uses all of this memory) starting at address 4TB. Here is another nice drawing from ZGC's source:

```
+--------------------------------+ 0x0000140000000000 (20TB)
|         Remapped View          |
+--------------------------------+ 0x0000100000000000 (16TB)
|     (Reserved, but unused)     |
+--------------------------------+ 0x00000c0000000000 (12TB)
|         Marked1 View           |
```

```
+--------------------------------+ 0x0000080000000000 (8TB)
|         Marked0 View           |
+--------------------------------+ 0x0000040000000000 (4TB)
```

At any point of time only one of these 3 views is in use. So for debugging the unused views can be unmapped to better verify correctness.

Pages & Physical & Virtual Memory

Shenandoah separates the heap into a large number of equally-sized *regions*. An object usually does not span multiple regions, except for large objects that do not fit into a single region. Those large objects need to be allocated in multiple contiguous regions. I quite like this approach because it is so simple.

ZGC is quite similar to Shenandoah in this regard. In ZGC's parlance regions are called pages. The major difference to Shenandoah: Pages in ZGC can have different sizes (but always a multiple of 2MB on x64). There are 3 different page types in ZGC: *small* (2MB size), *medium* (32MB size) and *large* (some multiple of 2MB). Small objects (up to 256KB size) are allocated in small pages, medium-sized objects (up to 4MB) are allocated in medium pages. Objects larger than 4MB are allocated in large pages. Large pages can only store exactly one object, in constrast to small or medium pages. Somewhat confusingly large pages can actually be smaller than medium pages (e.g. for a large object with a size of 6MB).

Another nice property of ZGC is, that it also differentiates between *physical* and *virtual* memory. The idea behind this is that there usually is plenty of virtual memory available (always 4TB in ZGC) while physical memory is more scarce. Physical memory can be expanded up to the maximum heap size (set with `-Xmx` for the JVM), so this tends to be

much less than the 4 TB of virtual memory. Allocating a page of a certain size in ZGC means allocating both physical and virtual memory. With ZGC the physical memory doesn't need to be contiguous - only the virtual memory space. So why is this actually a nice property?

Allocating a contiguous range of virtual memory should be easy, since we usually have more than enough of it. But it is quite easy to imagine a situation where we have 3 free pages with size 2MB somewhere in the physical memory, but we need 6MB of contiguous memory for a large object allocation. There is enough free physical memory but unfortunately this memory is non-contiguous. ZGC is able to map this non-contiguous physical pages to a single contiguous virtual memory space. If this wasn't possible, we would have run out of memory.

Marking & Relocating objects

A collection is split into two major phases: marking & relocating. (Actually there are more than those two phases but see the source for more details).

A GC cycle starts with the marking phase, which marks all reachable objects. At the end of this phase we know which objects are still alive and which are garbage. ZGC stores this information in the so called live map for each page. A live map is a bitmap that stores whether the object at the given index is strongly-reachable and/or final-reachable (for objects with a `finalize` -method).

During the marking-phase the load-barrier in application-threads pushes unmarked references into a thread-local marking buffer. As soon as this buffer is full, the GC threads can take ownership of this buffer and recursively traverse all reachable objects from this buffer. Marking in an application thread just pushes the reference into a

buffer, the GC threads are responsible for walking the object graph and updating the live map.

After marking ZGC needs to relocate all live objects in the relocation set. The relocation set is a set of pages, that were chosen to be evacuated based on some criteria after marking (e.g. those page with the most amount of garbage). An object is either relocated by a GC thread or an application thread (again through the load-barrier). ZGC allocates a forwarding table for each page in the relocation set. The forwarding table is basically a hash map that stores the address an object has been relocated to (if the object has already been relocated).

The advantage with ZGC's approach is that we only need to allocate space for the forwarding pointer for pages in the relocation set. Shenandoah in comparison stores the forwarding pointer in the object itself for each and every object, which has some memory overhead.

The GC threads walk over the live objects in the relocation set and relocate all those objects that haven't been relocated yet. It could even happen that an application thread and a GC thread try to relocate the same object at the same time, in this case the first thread to relocate the object wins. ZGC uses an atomic CAS-operation to determine a winner.

While not marking the load-barrier relocates or remaps all references loaded from the heap. That ensure that every new reference the mutator sees, already points to the newest copy of an object. Remapping an object means looking up the new object address in the forwarding table.

The relocation phase is finished as soon as the GC threads are finished walking the relocation set. Although that means all objects have been relocated, there will generally still be references into the relocation set, that need to be *remapped* to their new

addresses. These reference will then be healed by trapping load-barriers or if this doesn't happen soon enough by the next marking cycle. That means marking also needs to inspect the forward table to *remap* (but not relocate - all objects are guaranteed to be relocated) objects to their new addresses.

This also explains why there are two marking bits (`marked0` and `marked1`) in an object reference. The marking phase alternates between the `marked0` and `marked1` bit. After the relocation phase there may still be references that haven't been `remapped` and thus have still the bit from the last marking cycle set. If the new marking phase would use the same marking bit, the load-barrier would detect this reference as already marked.

Load-Barrier

ZGC needs a so called load-barrier (also referred to as read-barrier) when reading a reference from the heap. We need to insert this load-barrier each time the Java program accesses a field of object type, e.g. `obj.field` . Accessing fields of some other primitive type do not need a barrier, e.g. `obj.anInt` or `obj.anDouble` . ZGC doesn't need store/write-barriers for `obj.field = someValue` .

Depending on the stage the GC is currently in (stored in the global variable ZGlobalPhase), the barrier either marks the object or relocates it if the reference isn't already marked or *remapped*.

The global variables ZAddressGoodMask and ZAddressBadMask store the mask that determines if a reference is already considered good (that means already marked or remapped/relocated) or if there is still some action necessary. These variables are only changed at the start of marking- and relocation-phase and both at the same time. This table from ZGC's source gives a nice overview in which state these masks can be:

	GoodMask	BadMask	WeakGoodMask	WeakBadMask
Marked0	001	110	101	010
Marked1	010	101	110	001
Remapped	100	011	100	011

Assembly code for the barrier can be seen in the MacroAssembler for x64, I will only show some pseudo assembly code for this barrier:

```
mov rax, [r10 + some_field_offset]
test rax, [address of ZAddressBadMask]
jnz load_barrier_mark_or_relocate

# otherwise reference in rax is considered good
```

The first assembly instruction reads a reference from the heap: `r10` stores the object reference and `some_field_offset` is some constant field offset. The loaded reference is stored in the `rax` register. This reference is then tested (this is just an bitwise-and) against the current bad mask. Synchronization isn't necessary here since `ZAddressBadMask` only gets updated when the world is stopped. If the result is non-zero, we need to execute the barrier. The barrier needs to either mark or relocate the object depending on which GC phase we are currently in. After this action it needs to update the reference stored in `r10 + some_field_offset` with the good reference. This is necessary such that subsequent loads from this field return a good reference. Since we might need to update the reference-address, we need to use two registers `r10` and `rax` for the loaded reference and the objects address. The good reference also needs

to be stored into register `rax`, such that execution can continue just as when we would have loaded a good reference.

Since every single reference needs to be marked or relocated, throughput is likely to decrease right after starting a marking- or relocation-phase. This should get better quite fast when most references are healed.

Stop-the-World Pauses

ZGC doesn't get rid of stop-the-world pauses completely. The collector needs pauses when starting marking, ending marking and starting relocation. But this pauses are usually quite short - only a few milliseconds.

When starting marking ZGC traverses all thread stacks to mark the applications root set. The root set is the set of object references from where traversing the object graph starts. It usually consists of local and global variables, but also other internal VM structures (e.g. JNI handles).

Another pause is required when ending the marking phase. In this pause the GC needs to empty and traverse all thread-local marking buffers. Since the GC could discover a large unmarked sub-graph this could take longer. ZGC tries to avoid this by stopping the end of marking phase after 1 millisecond. It returns into the concurrent marking phase until the whole graph is traversed, then the end of marking phase can be started again.

Starting relocation phase pauses the application again. This phase is quite similar to starting marking, with the difference that this phase relocates the objects in the root set.

Conclusion

I hope I could give a short introduction into ZGC. I certainly couldn't describe every detail about this GC in a single blog post. If you need more information, ZGC is open-source, so it is possible to study the whole implementation.

Back to posts