Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK

ALBERT MINGKUN YANG, Oracle
TOBIAS WRIGSTAD, Uppsala University

ZGC is a modern, non-generational, region-based, mostly concurrent, parallel, mark-evacuate collector recently added to OpenJDK. It aims at having GC pauses that do not grow as the heap size increases, offering low latency even with large heap sizes. The ZGC C++ source code is readily accessible in the OpenJDK repository, but reading it (25 KLOC) can be very intimidating, and one might easily get lost in low-level implementation details, obscuring the key concepts. To make the ZGC algorithm more approachable, this work provides a thorough description on a high-level, focusing on the overall design with moderate implementation details. To explain the concurrency aspects, we provide a SPIN model that allows studying races between mutators and GC threads, and how they are resolved in ZGC. Such a model is not only useful for learning the current design (offering a deterministic and interactive experience) but also beneficial for prototyping new ideas and extensions. Our hope is that our detailed description and the SPIN model will enable the use of ZGC as a building block for future GC research, and research ideas implemented on top of it could even be adopted in the industry more readily, bridging the gap between academia and industry in the context of GC research.

CCS Concepts: • **Software and its engineering → Garbage collection**; **Model checking;**

Additional Key Words and Phrases: Garbage collection, ZGC, SPIN, model checking

ACM Reference format:
Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 22 (August 2022), 34 pages.
https://doi.org/10.1145/3538532

1 INTRODUCTION

Traditional **stop-the-world** (**STW**) collectors in a JVM offer high application throughput by incurring little synchronization with application threads (also known as mutators). However, GC pauses in STW collectors often become large with large live sets or heap sizes, which make applications unresponsive. Detlefs et al. [2004] proposed G1GC to address this issue with marginal throughput reduction, compared with STW collectors. G1 allows users to specify a *soft real-time goal* (often ~100 ms) for GC pauses, and G1 meets this goal with high probability. More recently, two more collectors (ZGC [Lidén and Karlsson 2018a] and Shenandoah [Christine H. Flood 2014;

Albert Mingkun Yang most of this work carried out while at Uppsala University.
Parts of this work are supported by the Swedish Foundation for Strategic Research, grant SM19-0059, and donations from Oracle.
Authors' addresses: A. M. Yang, Oracle, Söder Mälarstrand 27b, Stockholm, SE-118 25, Sweden; email: albert.m.yang@oracle.com; T. Wrigstad, Information Technology, Uppsala University, Lägerhyddsvägen 2, Uppsala, SE-751 05, Sweden; email: tobias.wrigstad@it.uu.se.

0164-0925/2022/08-ART22 $15.00
https://doi.org/10.1145/3538532

Flood et al. 2016]) were added to OpenJDK to support applications with service guarantees of ~10 ms. The two new collectors move more work outside GC pauses, which improves latency at the cost of further throughput drops. The focus of this article is the ZGC collector, from Oracle.

While technical talks and blog posts exist that describe various aspects of the ZGC algorithm, there are no scholarly works explaining the algorithm and the key aspects of its implementation thoroughly. The ZGC implementation in OpenJDK is of interest to researchers wanting to conduct memory management research in the context of a production-grade, modern, concurrent collector for a mainstream programming language. The ZGC code is sufficiently isolated from the rest of OpenJDK to allow it to be understood and extended without changes propagating through the entire system. In the past, we have ourselves built on ZGC [Yang et al. 2020b, a], and been forced to piece together information from a large number of sources, expressed slightly differently. Without direct access to Oracle engineers and eventually contributors, our work would have been considerably harder.

Therefore, to aid researchers like ourselves and to promote memory management research on OpenJDK, this article constitutes a deep dive into the ZGC algorithm, from its high-level concepts, key abstractions, key invariants, and design decisions to low-level implementation details. We take special care to focus on concurrent reference processing, which is subtle and difficult to understand from simply reading the code. We also provide a SPIN model as a starting point for exploring the interactions of design decisions with the ZGC correctness invariants.

The article makes the following contributions.

— *Detailed description of ZGC in natural language*: We provide the first thorough description of the implementation of ZGC in OpenJDK in a scholarly work, starting from a high-level overview and some preliminary benchmarking results (Section 2), to a detailed and in-depth explanation of the internals with only strong references (Section 3). Finally, showing that the presence of non-strong references (all four kinds, three visible at the language level and one internal to the JVM) (Section 4) increases the complexity drastically, and how ZGC addresses unique challenges in concurrent reference processing (Section 5).

— *ZGC description through a SPIN model*: Since understanding the concurrent interaction between mutators and/or GC threads is key to comprehending a modern concurrent GC algorithm, we provide a SPIN (Section 6) model of ZGC that extracts the key concurrent behaviors, which are spread over 25 KLOC in C++, into a single coherent SPIN model of about 900 LOC. The model provides an interactive experience for understanding ZGC and some of its most important invariants (Section 6.3). With the help of SPIN, one can comprehend why the corresponding code in ZGC is written that way by studying these invariants. An effective way of approaching an unfamiliar system is to deliberately break an invariant and see how it fails. Because of how model checking works, when something goes wrong in a highly concurrent system, SPIN reports the failure in a deterministic and timely fashion, as opposed to the usual non-deterministic debugging experience.

2 ZGC PERFORMANCE OVERVIEW

ZGC is a non-generational, region-based, mostly concurrent, parallel, mark-evacuate garbage collection algorithm implemented in OpenJDK for 64-bit architectures. Non-generational pertains to each GC cycle involves marking all live objects in the whole heap. Region-based pertains to heaps being divided into regions of different sizes governed by certain size classes. Mostly concurrent pertains to the fact that almost all GC work, marking, and heap defragmentation, are performed while mutators are running except for brief STW pauses for synchronization. Parallel pertains to the utilization of multiple threads for GC work. Finally, mark-evacuate pertains to heap

defragmentation by evacuating live objects from sparse regions to new regions and reclaiming those sparse regions.

For clarity, this article describes the production-ready version of ZGC released in OpenJDK 15. Most details also apply to experimental releases available since OpenJDK 11.

A major challenge concurrent collectors have to address is how to ensure that mutators and GC threads share a coherent view of the heap. Without special treatment, mutators may mutate the object graph in a way that invalidates a decision the GC just made. A typical solution to this problem is to have mutators run some extra code when objects are read (read barrier) or written (write barrier) so that the GC is informed and can act accordingly. The ZGC algorithm uses a special kind of read barrier, called a load barrier (explained in more detail in Section 3.1), to ensure mutators and GC share a coherent view. During marking, ZGC implements the incremental-update algorithm [Pirinen 1998; Wilson 1992], maintaining the strong tricolor invariant [Dijkstra et al. 1978], i.e., there are no pointers from black objects to white objects. After marking, sparsely populated regions are selected for evacuation. During the evacuation, load barriers are again used to maintain the invariant that mutators never see pointers into regions of memory marked for evacuation. Any such pointer will be updated in the load barrier to point to an object's new location, possibly including performing the relocation, in competition with GC threads. This concept is very similar to Baker-style relocation [Baker 1978]. Because relocation information is stored outside the original regions, old regions can be reclaimed after evacuation, which concludes a ZGC cycle.

The ZGC design strives to deliver a max pause time of a few milliseconds with marginal throughput loss. To understand the performance implications of this design, we ran SPECjbb2015 [SPEC 2015] with different collectors in OpenJDK. SPECjbb2015 is the latest version of the Java business benchmark from **Standard Performance Evaluation Corporation** (**SPEC**) that measures throughput and latency with graduated load levels. The JVM is built from OpenJDK 15[1] using GCC 9.2. We ran SPECjbb2015® V1.03 (composite mode) on an AMD EPYC 7742 with 1 NUMA node, 16 cores per NUMA node (2 hyper-threads/core), 256 GB RAM, 1 MB L1, 8 MB L2, 64 MB L3, running GNU/Linux 5.4 on top of KVM. The JVM flags we used are `-Xms`⟨*X*⟩`G -Xmx`⟨*X*⟩`G`, setting the initial and max heap size to the same, `-XX:+UseLargePages`, using OS pages of 2 M size,[2] `-XX:+AlwaysPreTouch`, pretouching the heap at JVM startup,[3] and `-XX:Use`⟨*X*⟩`GC`, selecting a particular GC from G1, Parallel, Shenandoah,[4] and Z. No GC-specific tuning is performed, meaning the results represent out-of-the-box performance. We evaluated different heap sizes, 32 G, 64 G, and 128 G, for each collector. We did not go for smaller heap sizes because Shenandoah and Z report a large number of allocation stalls, essentially degrading to STW GCs. This is probably due to the fact that both are non-generational and concurrent GCs usually require larger headroom than STW GCs. Each configuration (heap size plus the collector) is run 5 times, which is enough to yield a non-overlapping confidence interval. Each run produces two jOPS scores: `max-jOPS` is a pure throughput metric and `critical-jOPS` is a metric that measures critical throughput under **service level agreements** (**SLAs**) specifying response times ranging from 10 ms to 100 ms.

In Figure 1, the average and 95% **confidence interval** (**CI**) of the SPECjbb jOPS is shown in the top[5]; for each heap size, the relative difference against G1 (the default collector) is shown in

[1]The corresponding source is available at https://github.com/openjdk/jdk/releases/tag/jdk-15+36.

[2]By default, OS pages are 4K in size, which will put much pressure on the TLB for large heaps. Therefore, we pre-allocate some 2M-size pages and specify that the JVM uses them in order to reduce TLB misses.

[3]By default, physical memory is allocated only when unmapped virtual memory is accessed. Pretouching the heap will force the OS to allocate physical memory for the whole heap, thus avoiding page faults and potential latency hiccups during the run.

[4]Shenandoah used to use a Brooks-style barrier but switched to a Baker-style barrier since OpenJDK 13 [Kennke 2019].

[5]These graphs are *not* continuous functions; the dashed joining line is for visualization purposes only.

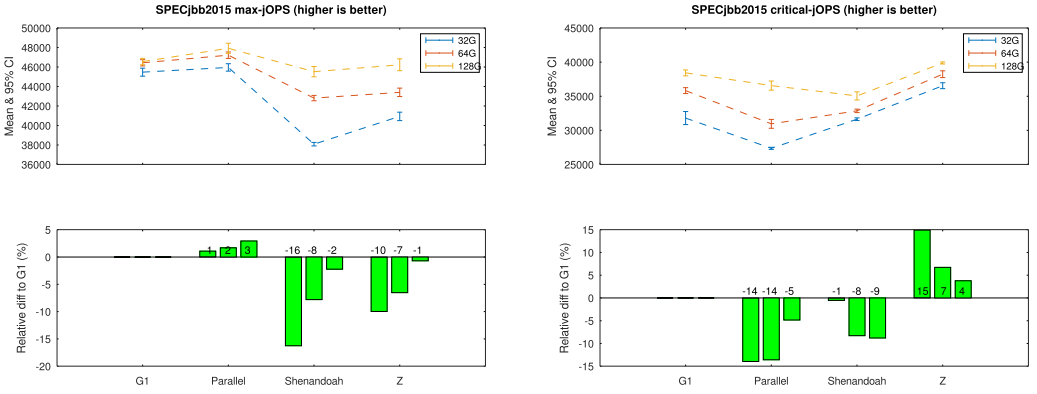


Fig. 1. Performance overview: SPECjbb2015 max-jOPS (left) and critical-jOPS (right) scores, using different GCs and heap sizes, AMD system.

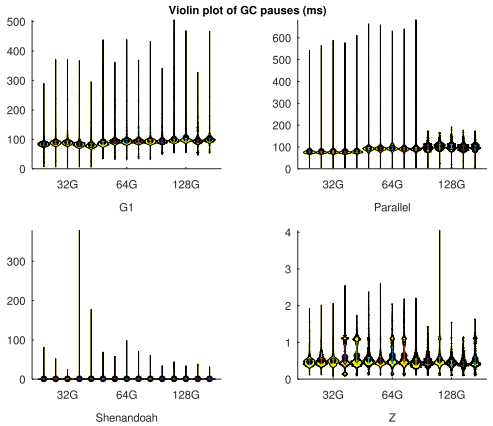


Fig. 2. Violin plots of GC pauses, using different GCs and heap sizes, AMD system.

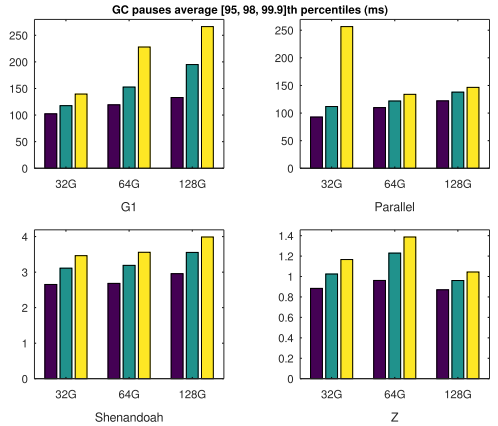


Fig. 3. Average percentiles of GC pauses, using different GCs and heap sizes, AMD system.

the bottom. Compared with G1, ZGC exhibits some regression in throughput (−10%) and some improvement in latency (15%) for 32 G heap size, but the difference shrinks as the heap size increases.

We used the JVM built-in logging infrastructure to emit GC events and extracted the GC pause times from the generated logs.[6] The violin plots of the GC pause time for each collector are shown in Figure 2.[7] The *x*-axis list all 15 runs (five runs for each heap-size). One can see that the GC pause time is significantly improved: most pause times are ~1 ms for Z, in contrast to ~100 ms for G1. In order to understand the significance of the long tails quantitatively, we also plot the 95th, 98th, and 99.9th percentiles, averaged over different runs, as shown in Figure 3. This result shows that ZGC is favorable in scenarios where better latency is desirable with modest throughput loss.

We repeated the same experiment on an Intel® Xeon® CPU E5-2630 @ 2.20 GHz with 2 NUMA nodes, 10 cores per NUMA node (2 hyper-threads/core), 256 GB RAM, 64 KB L1, 256 KB L2,

[6]We ignore pause time introduced by explicit `System.gc()`, since such pauses are initiated by applications as part of the benchmarking infrastructure.

[7]For Shenandoah, the majority of GC pauses are ~1 ms, but the large variance makes that less visible, so a zoom-in view of pause time for Shenandoah is provided in Figure 30 in the Appendix.

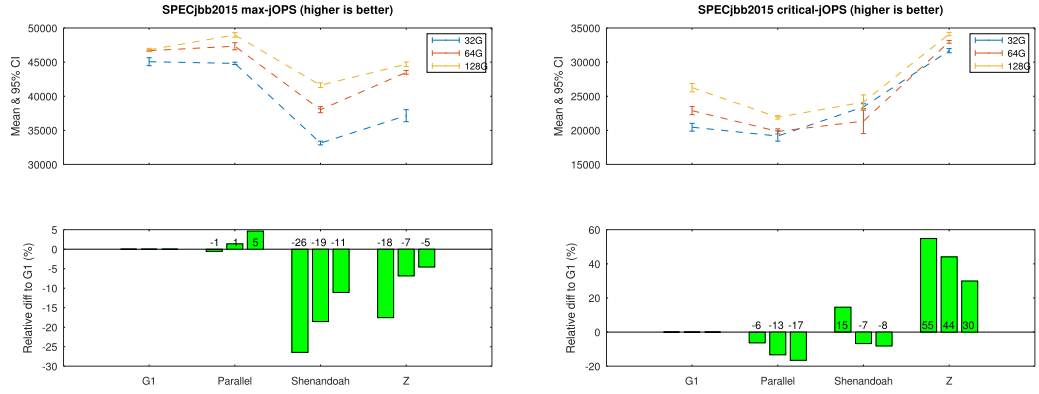


Fig. 4. Performance overview: SPECjbb2015 max-jOPS (left) and critical-jOPS (right) scores, using different GCs and heap sizes, Intel system.

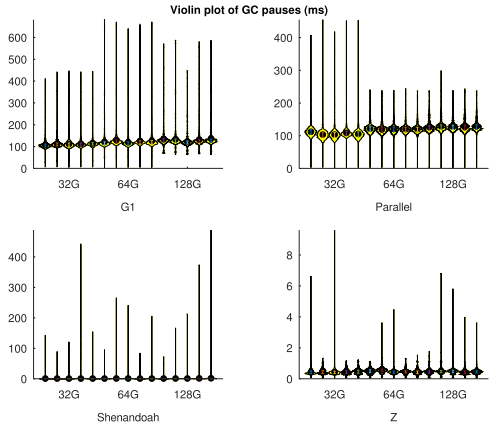


Fig. 5. Violin plots of GC pauses, using different GCs and heap sizes, Intel system.

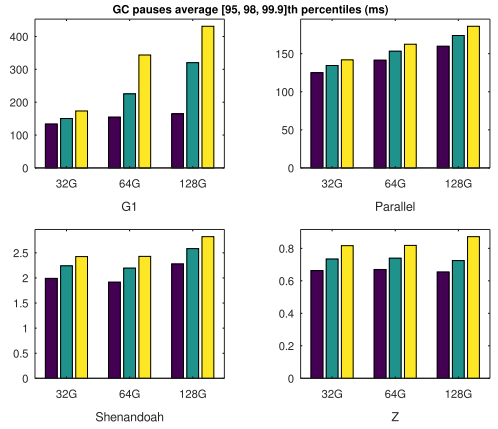


Fig. 6. Average percentiles of GC pauses, using different GCs and heap sizes, Intel system.

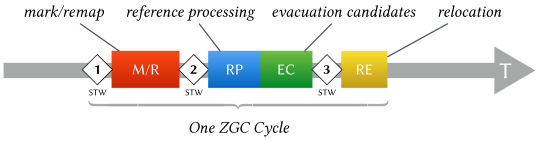
25 MB L3, running GNU/Linux 5.4. Because this box has 2 NUMA nodes, we used one more JVM flag, `-XX:+UseNUMA`, to enable the collector-specific NUMA support. The jOPS scores are shown in Figure 4, and GC pause time in Figures 5 and 6.[8] Larger difference against G1 can be observed (worse regression in throughput, but better improvement in latency), but the general trend and the conclusion are the same: the difference between G1 and ZGC shrinks as the heap size increases, and ZGC improves GC pause time significantly with modest throughput loss.

Having had a brief look at the performance of ZGC, we move deeper into its internals to study its design and implementation.

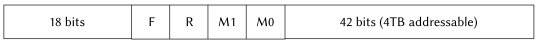
3 ZGC WALK-THROUGH

ZGC departs significantly from previous GC algorithms in the HotSpot JVM by utilizing load barriers instead of write barriers. Load barriers facilitate concurrent relocation: during relocation, an object may be moved at any time without updating its incoming pointers (which may exist anywhere in the heap), which effectively produces dangling pointers. However, load barriers trap

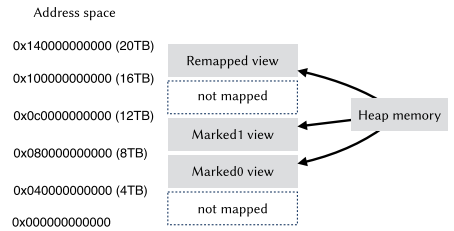
[8]For the same reason as footnote 6, a zoom-in view of pause time for Shenandoah is provided in Figure 31 in the Appendix.



(a) Phases and pauses in a ZGC Cycle.



(b) Meta (color) and address bits of a pointer.



(c) Multiple mapping of heap memory.

Fig. 7. Phases in a ZGC Cycle, pointer structure, and memory mapping.

loads of such dangling pointers and trigger code that updates the pointers with the new locations of relocated objects, thereby "fixing" the dangling pointers.

The trapping of dangling pointers is achieved by embedding metadata in pointers, using higher-order bits in addresses, similar to Pauseless GC [Click et al. 2005]. ZGC denotes such metadata as "colors", and all pointers mediate between two colors: good (pointer is valid) and bad (pointer is potentially invalid). Instead of all threads agreeing on the authoritative location of all objects atomically—which would impact the efficiency of concurrent relocation—threads must agree on what is the current good color, which is achieved via an STW pause. To keep track of how objects move so that dangling pointers can be fixed on load, forwarding tables are used to map pre-relocation (old) to post-relocation (new) addresses. Dangling pointers due to relocation will be fixed, either in the load barrier by mutators as a side-effect of accessing those pointers or by GC threads traversing all live objects in the heap (during marking). Thus, right after marking, a ZGC heap is guaranteed to have no dangling pointers.

The remainder of this section explains the design and implementation of ZGC in detail. The meat of this section mirrors the ZGC cycle, shown in Figure 7(a). The ZGC cycle consists of three STW pauses and four concurrent phases: **marking/remapping** (**M/R**), **reference processing** (**RP**), selection of **evacuation candidates** (**EC**), and **relocation** (**RE**). Before we dive into each phase of the GC cycle, we discuss two of ZGC's key design elements mentioned above: colored pointers and load barriers. Finally, we end with an example illustrating a full ZGC cycle.

3.1 Colored Pointers and Load Barriers

To ensure that mutators see only valid pointers, even with GC running concurrently, ZGC utilizes colored pointers and load barriers. Pointers are always 64-bit structures, consisting of meta bits (color of the pointer) and address bits. The number of address bits determines the theoretical maximal heap size supported. Heap space is broken into memory regions of one of three size classes, small, medium, and large. These regions are called pages inside OpenJDK, and we adopt this terminology.[9] Depending on its size, an object is allocated on a page of one particular size class using bump-pointer allocation. Pages of small and medium size classes can accommodate multiple objects, but pages of large size classes hold only a single object. In other words, an object allocated on a large page gets its own page.[10]

[9]To avoid confusion with OS pages, we will explicitly prefix page by OS when we mean an OS page from this point on.
[10]This is to avoid relocating large objects, which entails copying a large memory range and could incur high latency for mutators, see more in Sections 3.6 and 3.8.

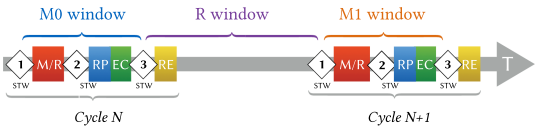


Fig. 8. The spanning window of good colors.

As shown in Figure 7(b), the lower 42 bits are address bits, the middle 4 bits are meta bits, and the higher 18 bits are unused.[11] The four meta bits are **Finalizable** (**F**), **Remapped** (**R**), **Marked1** (**M1**), and **Marked0** (**M0**). A pointer's color is determined by the status of its meta bits: F, R, M1, and M0. A color can be either "good" or "bad". A good color is one of the R, M1, M0 meta bits set and the other three unset, which gives three good colors: 0100, 0010, and 0001. At any instant of time, there is a globally agreed-upon single good color, and its selection is decided twice during a ZGC cycle (Figure 8): in STW1, where it alternates between M1 (0010) or M0 (0001) set, and in STW3, where it equals R being set, 0100. Once the good color is decided, all other colors are considered bad. Object creation always yields a pointer with the current good color.

The same physical heap memory is mapped into the virtual address space three times, resulting in three "views" of the same physical memory; each view corresponds to one good color as illustrated in Figure 7(c) so that pointers with good color can be dereferenced directly, and the corresponding virtual-to-physical memory mapping will be used. Only the view corresponding to the *current* good color is active and accessed. The underlying technique for maintaining multiple views into the same physical address is very similar to `Map2` in [Ossia et al. 2004] and `DoubleMap` in Compressor [Kermany and Petrank 2006].

A read barrier is code executed when reading a pointer from the heap, e.g., in `var x = obj.field`, where `x` is a local variable living on the stack, and `field` is a pointer living on the heap.[12] If the barrier is invoked on `field`, it is called a *load barrier*, ensuring only "clean" pointers live on the stack. In contrast, if the barrier is invoked on `obj`, it is called a *use barrier*, ensuring pointers are "cleaned" before using (dereferencing). ZGC uses load barriers to ensure the pointer in `field` has the good color. If the color of the pointer being loaded is good, the fast path of the load barrier is taken, otherwise, the slow path. The fast path is effectively empty, while the slow path contains logic for calculating the corresponding pointer with good color: checking if the object has been (or is about to be) relocated and if so, looking up (or deciding) the new address of the object. As a side-effect of the load barrier, the field is *self-healed*, meaning the old pointer is replaced by the one with good color so that subsequent accesses take the fast path. The concept of self-healing is also used in Pauseless GC [Click et al. 2005]. Regardless of which path is taken, a pointer with good color is returned so that the memory pointed to by that pointer can be accessed using the active view, associated with the good color. The snippet in Figure 9 shows the skeleton of the barrier and self-healing logic. The `slot` pointer refers to the location of `addr` and is used for self-healing, which uses `CAS` (compare-and-swap) to be thread-safe.[13] The rationale for skipping `NULL` pointers (Line 4 of Figure 9(b)) will be covered in Section 6.3.2. The terminating condition is either `CAS` returns `true`, meaning the current thread successfully performed self-healing, or the pointer value in the slot

[11] This bits arrangement renders 4 TB theoretical maximal heap size. Additionally, 8 TB and 16 TB max heap sizes are supported as well by having more address bits and shifting meta bits to higher bits accordingly. All work in similar ways, so we focus only on the 4 TB case.

[12] Here we assume `field` is not of primitive type; in other words, it is a pointer.

[13] `CAS` has the same semantics as `atomic_compare_exchange_strong(volatile A* obj, C* expected, C desired)` in C11 [ISO/IEC 2011]: atomically compares the memory contents pointed to by `obj` and `expected`, and if they are equal, replaces the former with `desired` and returns `true`; otherwise, loads the actual memory content pointed to by `obj` into `*expected` and returns `false`.

```
uintptr_t barrier(uintptr_t *slot,
                  uintptr_t addr) {
  // fast path
  if (is_good_or_null(addr)) return addr;
  // slow path
  good_addr = ...
  // self heal
  self_heal(slot, addr, good_addr);
  return good_addr;
}
```

(a) Fast/slow path in the barrier logic.

```
void self_heal(uintptr_t *slot,
               uintptr_t old_addr
               uintptr_t new_addr) {
  if (new_addr == 0) return;
  while (true) {
    if (CAS(slot, &old_addr, new_addr))
      return;
    if (is_good_or_null(old_addr))
      return;
  }
}
```

(b) Self-healing to avoid hitting slow path again.

Fig. 9. Skeleton of the barrier logic (a) and self-healing (b).

is already of good color or `null` (`is_good_or_null(old_addr)`), meaning another thread managed to do self-healing or mutators have written some value to the slot. (In Section 5.5, we will see that pointers are not always self-healed to the current good color. The loop in `self_heal` ensures an already good-colored pointer will not be tinted with bad colors; otherwise, subsequent accesses will take the slow path, defeating the original purpose of having self-healing.)

We now continue by describing the STW pauses and concurrent phases of ZGC in the left-to-right order in Figure 7(a).

3.2 STW1: The Start of the ZGC Cycle

The start of the ZGC cycle is a STW pause, which performs three major tasks. *First*, all threads agree on the current good color, meaning either the M0 or M1 bit is set by alternating selection.

Second, fresh pages are created to replace the currently used allocating pages for mutators to use so that future allocations (after STW1) will be placed on those fresh pages. This separates pages allocated prior to the current GC cycle from those allocated in or after the current GC cycle. ZGC assumes all objects on those fresh pages are live for this GC cycle. In other words, ZGC collects garbage objects only on pages allocated prior to the current GC cycle; those pages are called *relocatable* pages, because objects on them may be relocated as part of defragmenting the heap. This way, ZGC never follows pointers into fresh pages, because objects on those pages are likely to be used by mutators, and loading them could cause unnecessary cache traffic. It is safe not to mark through objects on fresh pages, because any objects that are only referenced by objects on fresh pages must have been loaded by mutators, which caused them to be placed on the mark stack (more is covered in the explanation of the load barrier in Section 3.3).

Third, the mark barrier (sharing the same skeleton in Figure 9(a) with the load barrier and to be explained in the next section) is applied for roots, such as system classes and objects, references on the stack frame, and so on. The mark barrier will detect invalid pointers (meaning pointers with bad color), and self-heal them—updating their address if needed and tinting them with a good color. The reason why roots need to have good color is that mutators will not hit the load barrier when accessing roots (recall loading variables on the stack does not trigger the load barrier). In addition, the mark barrier also pushes roots onto a mark stack.

In summary, by the end of STW1:

(1) all threads agree on what the current good color is;
(2) pages allocated before this cycle are identified, and the current GC cycle collects only dead objects on those pages;
(3) all roots have a good color and are pushed to the mark stack for concurrent marking.

```
while (obj in mark_stack) {
  // update liveness info
  success = mark_obj(obj);
  if (success) {
    for (e in obj->ref_fields()) {
      MB(slot_of_e, e);
    }
  }
}
```

Fig. 10. Main GC marking loop: for each object in the marking stack, mark it and call mark barrier on its fields of reference type.

```
1 void MB(uintptr_t *slot,
2         uintptr_t addr) {
3   // only null will early return
4   if (is_null(addr)) return;
5   if (is_pointing_into(addr, EC)) {
6     good_addr = remap(addr);
7   } else {
8     good_addr = good_color(addr);
9   }
10  mark_stack->add(good_addr);
11  self_heal(slot, addr, good_addr);
12 }
```

(a) Mark barrier.

```
1 uintptr_t LB(uintptr_t *slot,
2              uintptr_t addr) {
3   if (is_good_or_null(addr)) return addr;
4   if (is_pointing_into(addr, EC)) {
5     good_addr = remap(addr);
6   } else {
7     good_addr = good_color(addr);
8   }
9   mark_stack->add(good_addr);
10  self_heal(slot, addr, good_addr);
11  return good_addr;
12 }
```

(b) Load barrier.

Fig. 11. Mark barrier and load barrier; MB is used by GC threads, while LB is used by mutators.

3.3 Marking/Remapping (M/R)

The marking/remapping phase commences after STW1. In this phase, GC threads consume the mark stack, mark the popped objects, and update the liveness information of the associated page. The liveness information is the number of live bytes on a page and is used to select pages on which objects will be evacuated, meaning they will be relocated as part of defragmenting the heap. This will be further explained in Section 3.6. The snippet in Figure 10 shows the main GC loop of the M/R phase. The function `mark_obj` returns true if and only if the object was not marked and the current thread successfully marked the object. It uses an atomic operation (CAS) internally to set bits in a bitmap, so it is thread-safe.[14] Finally, in the true case, the mark barrier (MB, Figure 11(a)) is applied to this object's fields of the reference type.

The content of the MB is shown in Figure 11(a). It uses the barrier skeleton from Figure 9(a). The slow path updates a pointer in bad color: if the pointer points into the evacuation candidate set (EC) from the previous GC (meaning that the object was relocated, see Section 3.8 for details), we remap it to the new location tinted with the good color; otherwise, we just change the meta bits to match the good color. In either case, the good colored pointer is pushed onto the mark stack, and written back into the slot. Figure 11(b) shows the load barrier, which is applied in mutators on loading pointers from the heap. The main logic closely resembles what is in the mark barrier.[15] Since all

[14] `mark_obj` is effectively a CAS loop, trying to set certain bits in a bitmap. We are not showing its code here, because the same logic exists in many concurrent/parallel collectors and its abstraction level is lower than other code snippets in the article.

[15] The name, "mark barrier", is kind of a misnomer—it is not really a barrier itself. However, we kept this name in order to be consistent with the actual ZGC source code, and consequently, one can see the similarity between mark barrier and load barrier in both their names and implementation structures.

roots are pushed to the mark stack in STW1 and the load barrier pushes the to-be-accessed pointer to the mark stack, one can see this is the increment-update technique (in contrast to snapshot-at-the-beginning) with black mutators, hence, maintaining the strong tricolor invariant.

There are two places where the MB differs from the original barrier skeleton. First, the return type is `void` because we are interested only in the side-effect. Second, even pointers with good color will hit the slow path. The reason is that mutators and GC threads have their own thread-local marking stacks, and we would like as many objects as possible to be placed on the GC mark stacks. Otherwise, GC threads may run out of work and tentatively believe marking will soon be finished, while mutators still have a large marking backlog. This is safe because items in mutators' thread-local mark stacks are processed by GC threads before declaring marking is done. However, processing items belonging to another thread always introduces certain synchronization costs. The drawback of forcing pointers with good color to hit the slow path is that a single pointer can be placed in multiple mark stacks, which is safe because `mark_obj(obj)` is thread-safe.

3.4 STW2: The End of the Marking Phase

The marking phase terminates when all objects are marked. Checking when this condition becomes true is non-trivial: not only does each mutator and GC thread have its own thread-local mark stack that keeps a backlog of objects to mark, but mutators are pushing new items onto these mark stacks while they are being consumed by GC threads concurrently. In order to get the conclusive status of all mark stacks, we check this condition inside an STW pause, STW2. We have reached the end of M/R if all mark stacks are empty; otherwise, we are still in the M/R phase, and all mutators can resume work while GC threads consume mark stacks. As a result of this, we may enter STW2 prematurely, even multiple times during a single cycle. This behavior is naturally undesirable, as entering an STW pause requires global synchronization among all mutators and thus reduces their throughput. Therefore, thread-local handshaking with each mutator (one mutator at a time) is performed to check for the presence of any to-be-marked objects before attempting an STW pause; this reduces the probability of entering STW2 prematurely.

3.5 Reference Processing (RP)

The reference processing phase handles Java's Soft, Weak, and Phantom references. This phase is quite involved, and also complicates the marking phase. Therefore, we describe the treatment of those references in its own section, Section 5. For now, assume only strong references exist.

3.6 Selection of Evacuation Candidates (EC)

The evacuation candidate set is a collection of sparsely populated relocatable pages. After relocating all *live* objects on the EC pages into other pages, all EC pages can be reclaimed.[16]

First, before selecting evacuation candidates, the EC set from the previous GC cycle is cleared, and forwarding tables (mapping old addresses to new) associated with those pages are dropped. This is safe as there can be no reachable pointers in the heap still pointing into these pages—these pointers have been remapped by mutators or GC threads by the end of the M/R phase (see Lines 4–5 and 10 in Figure 11(a)).

Second, relocatable pages (i.e., allocated prior to the current GC cycle) with at least one live object (`page->is_marked() == true`) are tentatively added to the EC set, and pages with no live objects are reclaimed right away. Then, we sort all pages in EC by live bytes. Finally, a cut-point is picked, subject to the fragmentation limit threshold, so that the trailing multiple pages (too many live objects) are dropped from the EC set; more details on this are covered in HCSGC [Yang et al. 2020a].

[16]Here we assume relocation always succeeds, see discussion about relocation failure in Section 3.8.

```
remove_all(candidates);
for (page in pages) {
  // ZGC keeps objects allocated during
  // current cycle live
  if (!page->is_relocatable()) continue;
  if (page->is_marked()) add(candidates, page)
  else release_page(page);
}
sort(candidates);
// Construct EC depending on fragmentation limit
```

Fig. 12. High-level EC selection.

Since pages of large size contain only one object, which is either live or dead, the EC set contains only pages of small and medium size classes. In other words, pages of large size do not participate in relocation, and large objects are never relocated. Figure 12 shows the high-level algorithm.

3.7 STW3: Transitioning to Relocation

In STW3, prior to **relocation** (**RE**), the good color is to have the R bit set (0100). A pointer of this color is guaranteed *not* to point to objects on EC pages. This change of good color effectively invalidates all pointers in the heap and will cause mutators to take the slow path on the first subsequent load. We now explain how ZGC uses this invalidation to maintain the invariant that mutators never see pointers to regions of memory marked for evacuation (EC pages). In STW3, while mutators are stopped, all roots are visited—if a root points into an EC page, the object will be relocated to a non-EC page. Roots that do not point into EC pages are simply tinted with a good color. In the end, all roots will have good color; in other words, no roots point into EC pages. The slow path of the load barrier logic is updated in Figure 13(b). Compared with the load barrier logic in the M0/M1 window, we can see that it mostly stays the same. Note that Line 4 is the same as in the M/R phase, but pages in the EC have changed to the ones selected in the EC phase. Notably, line 5 means that mutators, before accessing an object in the EC, will relocate it to a new page outside of the EC (`relocate()` is elaborated in Section 3.8).[17] Together with the processing of roots in STW3, this establishes an invariant that mutators never see pointers into EC.[18]

3.8 Relocation (RE)

After STW3, the system is ready to perform concurrent relocation. This happens by GC threads migrating all live objects in the EC, page by page. The main loop of the relocation phase is shown in Figure 14(a),[19] and uses `relocate()` which is defined in Figure 14(b).

Instead of storing forwarding information (old-to-new address mapping) inside an old copy of a relocated object, it is stored in forwarding tables that live outside of each page (off-heap memory).[20] Forwarding tables require additional memory in comparison to reusing the already available from-space. However, they allow a relocated page to be reclaimed as soon as objects on it are evacuated, instead of having to wait until the end of the M/R phase, where one can be sure that all incoming pointers to all relocated objects have been remapped. Considering that the M/R phase is usually the dominant one in a GC cycle, the approach of storing forwarding tables outside of relocatable pages maintains a lower average memory footprint at the cost of a higher peak memory footprint.

[17] Actually, the load barrier cannot call `relocate()` directly due to a concurrency issue, which we will revisit in Section 6.3.5.
[18] This is essentially the same as the to-space invariant in Baker-style relocation [Baker 1978].
[19] Actually, `free_page()` cannot be called directly due to a concurrency issue, which we will revisit in Section 6.3.5.
[20] The size of a forwarding table is roughly proportional to the *number* of objects on a page selected for relocation, and each object takes ~16 bytes.

```
1 uintptr_t LB(uintptr_t *slot,
2              uintptr_t addr) {
3   if (is_good_or_null(addr)) return addr;
4   if (is_pointing_into(addr, EC)) {
5     good_addr = remap(addr);
6   } else {
7     good_addr = good_color(addr);
8   }
9   mark_stack->add(good_addr);
10  self_heal(slot, addr, good_addr);
11  return good_addr;
12 }
```

(a) Load barrier logic in M0/M1 window (reprinted from Fig. 11b).

```
1 uintptr_t LB(uintptr_t* slot,
2              uintptr_t addr) {
3   if (is_good_or_null(addr)) return addr;
4   if (is_pointing_into(addr, EC)) {
5     good_addr = relocate(addr);
6   } else {
7     good_addr = good_color(addr);
8   }
9   self_heal(slot, address, good_addr);
10  return good_addr;
11 }
```

(b) Load barrier logic in the R window.

Fig. 13. The logic of load barrier in the M0/M1 window (as we covered in M/R phase Section 3.3) and the updated version for R window.

```
1 for (page in EC) {
2   for (obj in page) {
3     relocate(obj);
4   }
5   free_page(page);
6   // page can be used for new allocation
7 }
```

(a) Main GC loop in relocation (RE) phase: relocating all objects in EC, page by page.

```
1 uintptr_t relocate(uintptr_t obj) {
2   // forwarding table for this address
3   ft = forwarding_tables_get(obj);
4   if (ft->exist(obj))
5     return ft->get(obj);
6   new_obj = copy(obj)
7   // CAS inside; linearization point
8   if (ft->insert(obj, new_obj))
9     return new_obj;
10  // relocation contention
11  dealloc(new_obj);
12  return ft->get(obj);
13 }
```

(b) Thread-safe relocate function.

Fig. 14. Main GC loop in RE phase (a), and thread-safe relocation function for a single object (b).

Mutators that access an object concurrently with its relocation will hit the slow path of the load barrier, as the pointer's color will invariably be bad. In this case, mutators help the GC threads perform the relocation, potentially competing with other relocating threads accessing the same object. Such competitions are handled at insertion into the forwarding table, which is the linearization point. Threads that fail the CAS, as indicated by the return value of `ft->insert(obj, new_obj)` (L8), will read the new address from the forwarding table (L12). Once such an entry exists, mutators hitting the slow path need only to query the forwarding table, skipping the relocation.

Once all objects on EC pages are relocated, the current ZGC cycle terminates, but there may still be stale pointers in the heap pointing to objects in EC. Those pointers will be fixed either by the next mutator access or GC threads in the M/R phase of the next ZGC cycle. Because all live pointers will be pushed to the mark stack during marking—mutators cannot hide any live pointers due to the particular implementation of the load barrier, all live pointers will be fixed (remapped) by the end of the M/R phase. Therefore, after marking, the ZGC heap is guaranteed free of dangling pointers.

In the above discussion, we made a simplification that memory allocation for relocation (L6 in Figure 14(b)) always succeeds. As a result, no live objects reside on EC pages after evacuation, and all EC pages can be reclaimed at the end of the RE phase. When memory allocation failure does

occur for an object, the old address of the object will be used as the new address, and the same insertion-into-forwarding-table logic proceeds. Additionally, the EC page this object resides on is specially marked so that not-yet-evacuated objects on the same EC page will not be evacuated, and this EC page will not be reclaimed at the end of the RE phase. In practice, because concurrent collectors are often given some headroom in the heap, relocation failure is rare.

3.9 ZGC Cycle Example

To illustrate the ZGC algorithm, Figure 15 walks through all phases in an example, modulo non-strong reference processing, adapted from a presentation by Lidén and Karlsson [2018b]. The subfigures go through the phases of a cycle in order.

Figure 15(a) shows the initial state of the heap. In Figure 15(b), M0 is selected as the good color, and all root pointers have a good color. All roots are then pushed to the mark stack, which is consumed by GC threads during M/R, as shown in Figure 15(c). We draw the objects themselves with a good color to indicate that they are marked, even though only pointers have color.

In Figure 15(d), the page with the fewest live objects (the page in the middle) is selected as an evacuation candidate. Subsequently, in Figure 15(e), the good color is defined as R bit on, and all root pointers have been updated to the good color. If a root was pointing into EC, the corresponding object would be relocated, and the root pointer updated to the new address (with good color).

In Figure 15(f), objects in EC are relocated, and the address translation is recorded in a forwarding table associated with the evicted page. When the RE phase ends, the current GC cycle terminates as well, rendering the whole EC reclaimable as shown in Figure 15(g). There are still outdated pointers due to relocation in the heap, which are not fixed until the next GC M/R phase, unless mutators try to load them, in which case they will be fixed in the slow path of the load barrier.

Now, the next cycle starts: in Figure 15(h), M1 is selected as the good color (as we alternate between M0 and M1). In Figure 15(i), all outdated pointers are remapped by querying the associated forwarding table. Finally, in Figure 15(j), the forwarding tables associated with EC pages of the previous cycle are reclaimed, in preparation for the upcoming RE phase.

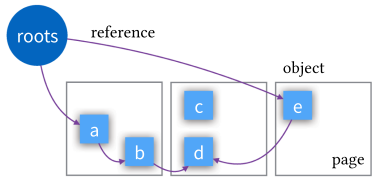
So far, we have explained how ZGC works with only strong references, i.e., references that keep their referents live (if the reference itself is live). Next, we will discuss different kinds of *non-strong references* and how they are used at the language level, and in ZGC. Readers who are familiar with them should feel free to skip directly to Section 5.

4 NON-STRONG REFERENCES

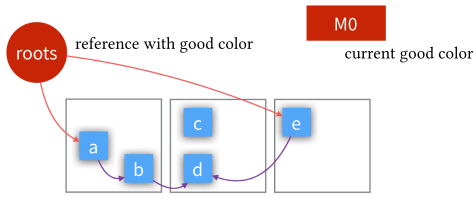
In addition to ordinary *strong* references created using the *new* operator, the Java language defines *Soft*, *Weak*, and *Phantom* references, which are collectively known as non-strong references. Non-strong references offer developers a limited form of interaction with the garbage collector as the JVM is allowed to collect objects which are reachable only via non-strong references.

As shown in Figure 16, non-strong references work like wrappers pointing to the actual object, called the *referent* of those references. To retrieve and clear the referent, Java provides the methods, `get()` and `clear()` respectively. Modulo clearing, references are immutable. Java's non-strong references introduce some additional refinement of reachability, shown in Table 1, adapted from the Java documentation of the package *java.lang.ref*.[21] (The row for Final reachability will be explained in Section 5.3; it is placed here for completeness.) A formal definition of different levels of reachability is provided in Ugawa et al. [2014]. Objects become collectible (unreachable)

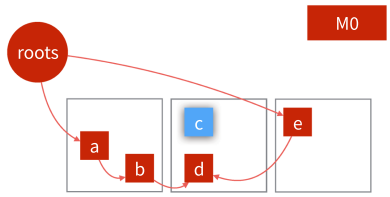
[21] See https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/ref/package-summary.html for more information about reachability.



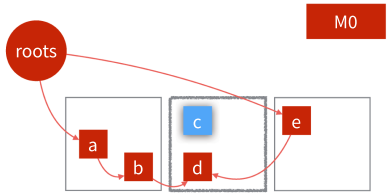
(a) Initial state. 4 live objects.



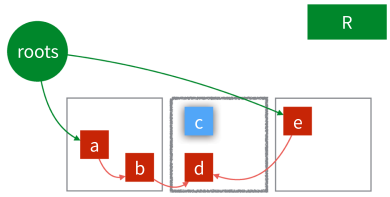
(b) STW1: roots have good color.



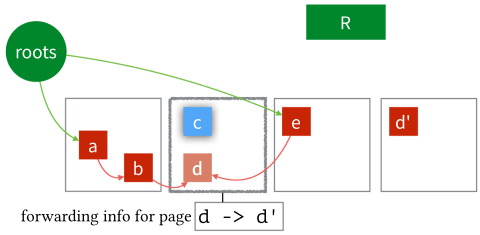
(c) M/R: all reachable objects are marked.



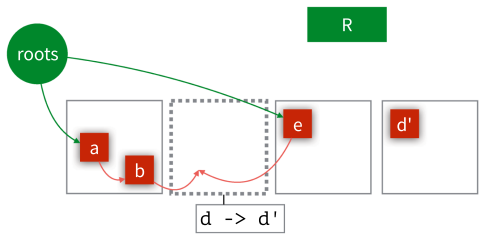
(d) EC: the page in the middle is selected as EC.



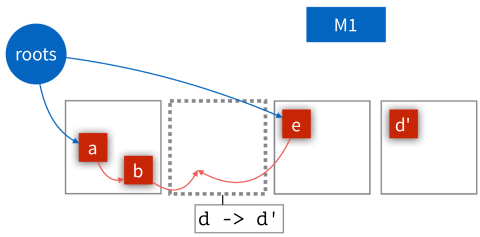
(e) STW3: roots have good color.



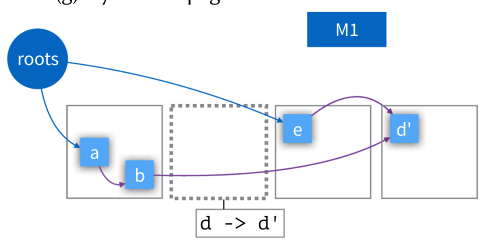
(f) RE: objects in EC relocated



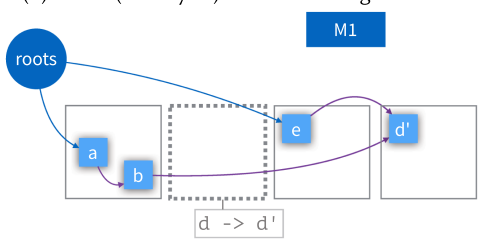
(g) Cycle end: pages in EC are reclaimed.



(h) STW1 (next cycle): all roots have good color.



(i) M/R (next cycle): all outdated pointers are remapped.



(j) EC (next cycle): forwarding tables from previous cycle are dropped.

Fig. 15. Example of a ZGC cycle. M/R = marking/remapping; EC = Evacuation Candidates; RE = relocation. Note that we draw objects in good color to indicate that they are marked, even though colors apply only to pointers.

Fig. 16. A non-strong reference and its referent. The pointer to the wrapper object is (in this case) strong, and the type of the wrapper determines the type of non-strong reference to its referent.

Table 1. Refined Reachability (Ordered from Strongest to Weakest) and Their Definitions

Reachability	Surface	Definition
Strong	Yes	Can be reached by some thread without traversing any reference objects
Soft	Yes	Not strongly reachable but can be reached by traversing one or more Soft references
Weak	Yes	Neither strongly nor softly reachable but can be reached by traversing one or more Weak references
Final	No	Neither strongly, softly, nor weakly reachable but can be reached by traversing one or more Final references
Phantom	Yes	Neither strongly, softly, nor weakly reachable, has been finalized and is reachable from one or more Phantom references

The Surface column indicates the presence of a language-level construct that exposes this feature to developers.

if they do not fit into any of the reachable categories. Note that by the definition of reachability in Table 1, an object always has one single reachability level, the strongest of all paths from roots to itself. For example, an object that has a "normal" strong reference to it, as well as a Weak reference, is strongly reachable. This means that the Weak reference will not be cleared. However, if the strong reference vanishes, the object becomes weakly reachable, and this Weak reference will be processed (cleared).

When the reachability of the referent corresponds to the type of the reference (e.g., a Weak reference wrapper's referent becomes weakly reachable as in the example above), reference processing, according to the Java Language Spec, provides two guarantees: (1) the reference will be cleared (its referent becomes `null`), and (2) if a queue is registered when the reference is constructed, the reference is added to the queue, which serves as a notification allowing developers to respond to the change of reachability. For examples of this, see Satish [2016].

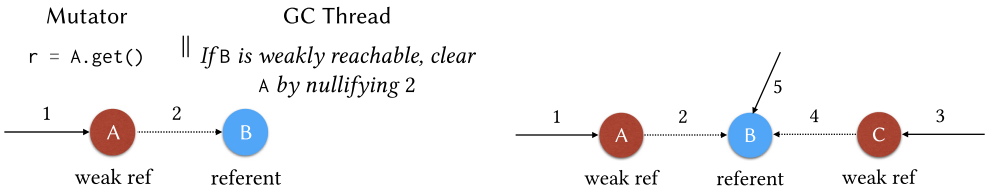
5 CONCURRENT REFERENCE PROCESSING

In this section, we describe the challenges of concurrent reference processing and continue by showing how they are addressed in ZGC.

5.1 Challenges of Concurrent Reference Processing

A mutator could call `get()` on the reference to upgrade the reachability of the referent to strongly reachable if the reference has not been cleared. Such upgrading of reachability means that an about-to-be-collected referent stops being eligible for collection; this process is called *resurrection* in the ZGC context.[22] Resurrection, performed by mutators, and clearing the reference, performed by GC threads as part of reference processing, could be racing with each other, which makes concurrent reference processing challenging.

[22] Object resurrection often refers to the instant when an object comes back to life during its destruction (inside the finalizer method); here we are using this term in a more general context.



(a) A mutator may resurrect an object right before the enclosing reference is cleared.

(b) Multiple references sharing the same referent must behave consistently.

Fig. 17. Two main challenges of concurrent reference processing.

There are two main challenges in supporting concurrent reference processing, which we will illustrate using two scenarios, graphically depicted in Figure 17(a) and (b).

The first challenge is the race between resurrection and clearing. In Figure 17(a), a mutator could call `a.get()` at the same time a GC thread decides to clear 2. The mutator's resurrection changes the reachability of the referent, thus invalidating the GC thread's decision.

The second challenge is to ensure consistent views across multiple references. In Figure 17(b), multiple references share a common referent, `b`. If `b` becomes *weakly reachable* (e.g., due to five vanishing), `a` and `c` must be cleared *at the same time*. Otherwise, resurrection could happen via the non-cleared reference, which would invalidate the clearing decision. In this example, it entails that `assert(a.get() == c.get())` must always hold.

Given these two challenges, it is not uncommon that many garbage collectors choose to perform reference processing in an STW pause, instead of concurrently. Next, we will show how concurrent reference processing works in ZGC, and how the two challenges above are addressed.

5.2 Overview of Concurrent Reference Processing in ZGC

The main logic of concurrent reference processing consists of five steps:

(1) identifying (discovering) all non-strong references which *potentially*[23] need to be processed and placing them on a ***discovered list*** (**DL**);
(2) disable resurrection, and while resurrection is blocked, `get()` will return the referent or `NULL` depending on the marking status of the referent;
(3) prune the discovered list using complete reachability information to filter out references that should not be processed, and process[24] all references left on the discovered list;
(4) re-enable resurrection, which closes the resurrection-blocked window;
(5) publish the discovered list so that its references can be placed in the registered queue (if any).

These five steps are distributed over the M/R, STW2, and RP phases as shown in Figure 18. Step 1 takes place during M/R because we need to traverse the whole heap to find all references that may need to be processed. Step 2 takes place during STW2 so that all mutators know that the resurrection-blocked window has been entered. After the M/R phase has terminated, we have the complete marking information, i.e., the final marking status for every object in the heap.

Within the resurrection-blocked window, the return value of `get()` depends on the marking status of the referent. Since resurrection becomes blocked in an STW pause, GC, and mutators share

[23]This is an over-approximation because mutators are running concurrently, which may affect the reachability of referents.
[24]The actual procedure differs for each reference kind, but it is fine to understand this step as clearing the referent for now. More is covered in Section 5.4.

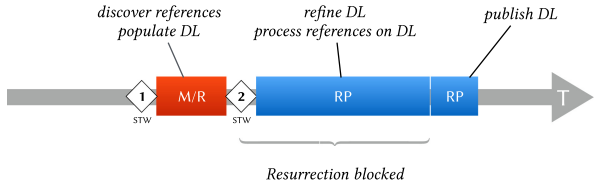


Fig. 18. Overview of concurrent reference processing in ZGC.

a consistent view of the reachability of the referent, addressing the first challenge; additionally, because all mutators share a consistent view, this also addresses the second challenge.

Steps 3–5 take place in the RP phase. Before diving into each step, we need to know a bit more about the internals of non-strong references.

5.3 Final Reachability

In addition to the three non-strong reference kinds visible in the Java language, there is one more kind of non-strong reference, called *Final* references, which are defined and constructed internally in the JVM (in both OpenJDK and Jikes). Whenever an object with a non-empty *finalize()*[25] method is created, an instance of a *Final* reference is created, and its referent is set to point to that object. In terms of reference strength, Final references are placed between Weak and Phantom references, because a weakly reachable object is eligible for finalization but has not been finalized, while a phantom reachable object has been finalized; hence the placement of the corresponding row in Table 1. Also, every Final reference is registered with a *final reference queue*; a Final reference will be enqueued when it becomes final reachable, just like other kinds of non-strong references. All final references are known to the VM and are roots of finalizable marking (explained in Section 5.5).

Next, we dive into the internal representation of all four non-strong references and explain how they are used for each kind in reference processing.

5.4 Non-strong Reference Internals

Internally in OpenJDK, non-strong references have three main fields:

(1) `referent`: the content of the reference as shown in Figure 16;
(2) `discovered`: used to form per GC-thread **discovered lists** (**DL**);
(3) `next`: used to form the registered queue.

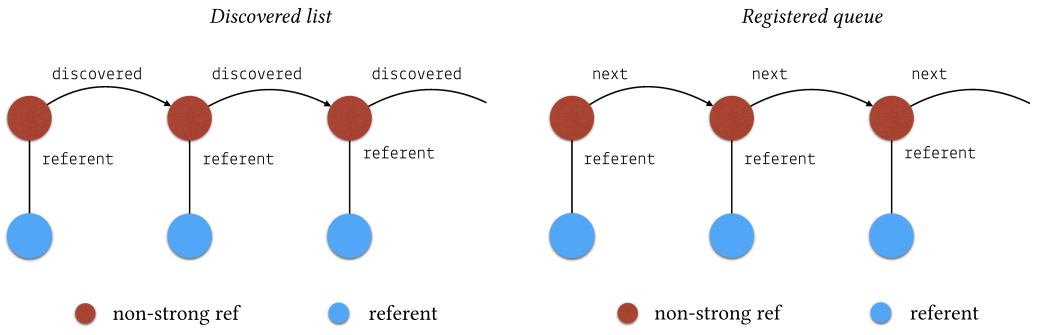
The content of the `referent` field can be retrieved via `get()` and cleared via `clear()`. The fields `discovered` and `next` can be understood as an optimization for creating a linked list structure without the need for additional memory allocation (sometimes called an intrusive list). The `discovered` field is used to chain references together into the discovered list, as shown in Figure 19(a), and is accessible only internally in the VM. The `next` field is used in a similar way when a reference is enqueued, either by mutators calling `enqueue()` or by the garbage collector, as shown in Figure 19(b).

Now that we are fully equipped with the knowledge of all reference kinds and their internal representation, we can revisit the story of reference processing outlined in Section 5.2, but in more detail.

5.5 Step 1: Discovery of Non-strong References in the M/R Phase

In the absence of non-strong references, while traversing the object graph recursively from the roots, every object encountered can be marked live. However, in the presence of the refined

[25]These are the destructor methods of Java and have been deprecated since Java 9.



(a) Discovered list chaining.

(b) Registered queue chaining.

Fig. 19. The discovered list and the registered queue are constructed by chaining references together using an internal field (`discovered` and `next` respectively).

```
1 while (obj in mark_stack) {
2   // update liveness info
3   success = mark_obj(obj);
4   if (success) {
5     for (e in obj->ref_fields()) {
6       MB(slot_of_e, e);
7     }
8   }
9 }
```

(a) GC marking loop (reprinted fromFig. 10).

```
1 while ((obj, finalizable) in mark_stack) {
2   success = mark_obj(obj, finalizable);
3   if (success) {
4     if (finalizable) {
5       for (e in obj->ref_fields()) {
6         FMB(slot_of_e, e);
7       }
8     } else {
9       if (obj->is_reference &&
10          should_discover(obj)) {
11        discover(obj);
12      } else {
13        for (e in obj->ref_fields()) {
14          MB(slot_of_e, e);
15        }
16      }
17    }
18  }
19 }
```

(b) Revised GC marking loop.

Fig. 20. Original (left) and revised GC marking loop with refined reachability (right).

reachability of Table 1, the binary marking status becomes insufficient to process all non-strong reference kinds in a single phase. It is therefore extended to three states:

(1) strongly-marked: strongly reachable;
(2) finalizably-marked: finalizably reachable;
(3) unmarked: unreachable, eligible for collection.

Both the original GC marking loop and the updated version are shown in Figure 20 to highlight the difference. In the updated version, each element in the mark stack contains an address of an object we would like to mark, as well as *how* it should be marked: strongly-marked or finalizably-marked, as shown on Line 1. It also affects how the recursive trace should proceed, finalizable-marking (Lines 5–7), or strong-marking (Lines 9–16). In the strong-marking case, non-strong reference discovery is performed to collect non-strong references that may need to be processed. Note

```
1 void MB(uintptr_t *slot,
2         uintptr_t addr) {
3   // only null will early return
4   if (is_null(addr)) return;
5   if (is_pointing_into(addr, EC)) {
6     good_addr = remap(addr);
7   } else {
8     good_addr = good_color(addr);
9   }
10  mark_stack->add(good_addr);
11  self_heal(slot, addr, good_addr);
12 }
```

(a) Mark barrier (reprinted from Fig. 11a).

```
1 void FMB(uintptr_t* slot,
2          uintptr_t addr) {
3   // never downgrading
4   if (is_good_or_null(addr)) return;
5   if (is_pointing_into(addr, EC)) {
6     good_addr = remap(addr);
7   } else {
8     good_addr = good_color(addr);
9   }
10  mark_stack->add(good_addr, finalizable);
11  fg_addr = FinalizableGood(good_addr);
12  // self heal to finalizable good instead of good
13  self_heal(slot, addr, fg_addr));
14 }
```

(b) Finalizable mark barrier.

Fig. 21. Original (left) and revised mark barrier supporting refined reachability (right).

Table 2. Discovery Conditions and Discovery Actions for Non-strong References

Ref. kind	Discovery Condition	Discovery Action
Soft	The referent is not strongly-marked and the soft reference processing policy concludes that this reference should be cleared (∗)	Add to discovered list
Weak	Referent is not strongly-marked	Add to discovered list
Final	Referent is not strongly-marked	Switch to finalizable marking (using FMB) for the referent and its recursively reachable closure, then, add to discovered list (†)
Phantom	Referent is not strongly-marked (‡)	Add to discovered list

that we do not perform such discovery in the finalizable-marking case because such references are not strongly reachable (i.e., not reachable from mutators).

As illustrated in Figure 21, the **finalizable mark barrier** (**FMB**) has two distinguishing deviations from the **mark barrier** (**MB**):

(1) If `addr` is already good, we do not perform finalizable-marking. This is not needed because strongly-marked objects are never downgraded to finalizably-marked (Line 4).
(2) Self-healing changes the color of a pointer to the finalizable-good color (good color *and* F bit set), instead of good color (Lines 11–13). This is needed to ensure that if a mutator later loads this pointer, the slow path will be taken, causing the corresponding object to be correctly strongly-marked.

During heap traversal, on visiting a non-strong reference, we will check if this reference should be discovered; if so, a discovery action is taken. Otherwise, it is treated as an ordinary object, and ordinary marking proceeds on its fields of reference type. Table 2 explains, for each kind of non-strong reference, in what situation it should be discovered (`should_discover(obj)`, and what the corresponding discovery action is (`discover(obj)`).

For the discovery condition of Soft references (∗ in Table 2), there are two policies available, controlled internally by ZGC: `AlwaysClearPolicy` and `LRUMaxHeapPolicy`. The former is used under high memory pressure, and causes all Soft references to be cleared, while the latter is used when memory pressure is low and clears a Soft reference only when it has not been used for a while. When `AlwaysClearPolicy` is used, Soft references behave like Weak references.

Table 3. Return Value of `get()` in Resurrection-blocked Window

	Non-Strong Reference Kind			
Referent's Marking Status	Soft	Weak	Final	Phantom
strongly-marked	referent	referent	n/a	`NULL`
finalizably-marked	`NULL`	`NULL`	n/a	`NULL`
unmarked	`NULL`	`NULL`	n/a	`NULL`

By the discovery action of Final references (†), finalizably-marked objects can be upgraded to strongly-marked, but strongly-marked objects cannot be downgraded.

The discovery condition of Phantom references (‡) may strike some astute readers as a surprise. If the referent is finalizably-marked, the reference should technically not be discovered/processed according to Table 1. Here, we conservatively push some Phantom references onto the discovered list to avoid their referents being strongly-marked, which would happen if the reference is not discovered as shown in Figure 20(b).

The discovered list is constructed along with marking, so the reachability information used is incomplete (a not-yet-marked or finalizably-marked referent could well become strongly reachable later on). As a result, the discovery list is actually an over-approximation for all kinds of references (not only Phantom references). Therefore, this list requires refinement using complete marking information to identify the accurate list of references that need to be processed.

5.6 Step 2: STW2 Revisited

As mentioned in Section 3.4, marking is finished when the mark stacks are empty. At this point we have the complete marking (reachability) status for all objects on relocatable pages: strongly-marked, finalizably-marked, or unmarked.

In addition, resurrection becomes blocked so that mutators cannot call `get()` on a non-strong reference to upgrade its referent's reachability. While resurrection is blocked, i.e., the resurrection blocked window in Figure 18, mutators calling `get()` will consult the marking status of the referent, and return the corresponding value as shown in Table 3. The column for Final references is inapplicable because Final references are not visible at the Java level. The all-`NULL` column for Phantom references is due to the fact that `get()` for Phantom is hard-coded to return `NULL` unconditionally.

5.7 Steps 3–5: Reference Processing

After the STW2 pause, the **reference processing** (**RP**) phase commences. In this phase, we use the complete marking information to select references that actually need to be processed (marking them non-active and adding them to the registered queue) in the discovered list.

We iterate over the discovered list, which consists of non-strong references collected during marking, and inspect each one of them to decide if it should be kept or dropped from the discovered list using the now complete marking information (as the M/R phase has completed). A Phantom reference should be dropped if its referent is marked at all (strongly-marked or finalizably-marked); a reference of any other reference kind should be dropped if and only if the referent is strongly-marked. Table 4 shows the corresponding actions for each non-strong reference type. After the filtering, all references on the discovered list should be processed.

Processing a reference means marking it non-active, so that calling `get()` returns `NULL`; otherwise, this would upgrade the reachability of the referent to strongly-reachable, contradicting the decision the GC just made, that the referent is not strongly-reachable anymore. Table 5 shows the action performed for marking a reference as non-active. For Soft, Weak, and Phantom references,

Table 4. Inspecting Each Non-strong Reference in the Discovered List and Deciding Whether it Should Be Kept or Dropped (Removed from the List) Depending on the Marking Status of the Referent

	Non-Strong Reference Kind			
Referent's Marking Status	Soft	Weak	Final	Phantom
strongly-marked	Drop	Drop	Drop	Drop
finalizably-marked	Keep	Keep	Keep	Drop
unmarked	Keep	Keep	—	Keep

Table 5. Corresponding Action Performed for Marking a Non-strong Reference as Non-active

Ref. Kind	Action
Soft	`ref.referent = NULL`
Weak	`ref.referent = NULL`
Final	`ref.next = ref`
Phantom	`ref.referent = NULL`

being marked as non-active means setting the `referent` to `NULL`.[26] For Final references, it means setting `next` to point to the reference itself. The special treatment for Final references is because the referent is needed to run the finalizer. Using the `next` field for this is possible because final references are not exposed at Java level, so the `next` field is not used by Java threads.

Now that all referents reflect their corresponding reachability using the complete marking information, resurrection can be safely re-enabled, since `get()` returns either `NULL` or objects that are strongly-marked; in other words, referents could be loaded directly via `get()` without querying the reachability information. Eventually, all per GC-thread discovered lists are merged into one *pending list*, which is published to the `ReferenceHandler` thread for further processing (adding to the registered queue). A final reference is always associated with the final reference queue at its creation, and an internal `FinalizerThread` polls the final reference queue regularly and invokes the finalizer on polled references. The work done after the pending list is published is outside of the scope of ZGC; it is implemented in Java code and shared by all garbage collectors in HotSpot.

5.8 Comparison with STW Reference Processing

Collectors that do not support concurrent reference processing often process Soft/Weak and Phantom references in different steps, separated by final-ref-marking. The STW reference processing logic mainly consists of four steps:

(1) strong-ref-marking: marking following only strong references;
(2) processing Soft/Weak references;
(3) final-ref-marking: marking following final and strong references;
(4) processing Phantom references.

The STW approach has two advantages: first, only a single marking bit is needed to store the marking status of each object; second, no overlap between strong-ref-marking and final-ref-marking: an object is either marked during strong-ref-marking or final-ref-marking, but never both. In contrast, ZGC requires two marking bits, and an object can be finalizably-marked first and become strongly-marked later.

Let us ponder whether it is possible to achieve concurrent reference processing by implementing steps 2–4 as concurrent (i.e., mutators are not paused) phases. (Step 1 is essentially marking, and G1 supports concurrent marking.) Here we illustrate one challenge if step 2 was to be implemented concurrently. This challenge stems from the presence of phantom-strength references but with *accessible* referents, i.e., `get()` does not unconditionally return `NULL`.[27] (Recall that referents of

[26]The treatment for Phantom references has changed since OpenJDK 9; see https://bugs.openjdk.java.net/browse/JDK-8071507 for details.
[27]`Reference.refersTo()` is added in JDK16 and it exposes the same problem as we outlined below. More about `refersTo()` at https://bugs.openjdk.java.net/browse/JDK-8241029.

Phantom references are always inaccessible, i.e., `get()` unconditionally returns `NULL`.) A reference of `jweak`[28] type, from **Java Native Interface** (**JNI**), is such an example.[29]

Let's revisit the STW approach, with `jweak` on the table. Suppose we have finished strong-ref-marking, and are in the middle of processing Soft/Weak references. How should `get()` behave for a `jweak` variable?[30] There are two questions to answer if the referent is not `NULL` and not marked (otherwise, one can just return the referent):

(1) What is the return value of `get()`, `NULL` or the referent object?
(2) Should the referent be marked?

For the first question, returning `NULL` would be problematic, because the referent could become marked in the upcoming final-ref-marking phase, and according to the reachability definition in Table 1, a phantom-strength reference should not be processed (cleared) if its referent is final reachable. Consequently, `get()` shall return the actual referent. Then, the answer to the second question becomes clear as well, because the returned referent must be marked in order to avoid dangling pointers—the mutator will have a dangling pointer after this unmarked object is reclaimed by GC. In summary, our preliminary reasoning suggests that `get()` for `jweak` should return the referent and mark the referent object if it is not `NULL`.

Unfortunately, such implementation can break reference processing atomicity: two Weak references sharing the same referent will be treated differently if the same referent was pointed by a `jweak` reference and a mutator calls `get()` on it—the first Weak reference will be processed because the referent is not marked, but the second Weak reference will *not* be processed because the referent is marked according to our proposed implementation of `get()` for `jweak`. Therefore, one cannot extend the STW reference processing algorithm to concurrent reference processing trivially.

In contrast, ZGC maintains the reference processing atomicity because the reachability of all live objects is completely decided in the M/R phase (before STW1). After STW1, this reachability information is used to decide whether/how a non-strong reference should be processed—mutators, no matter what they do, cannot alter reachability after STW1. With this comparison between STW and concurrent reference processing, we can better appreciate why ZGC performs strong-ref-marking and final-ref-marking in a single phase.

6 MODEL CHECKING USING SPIN

In order to understand a system completely, interacting with it is essential. Because our textual descriptions fall short in this regard, we constructed a SPIN model of ZGC. This model has many uses. First, it is useful for learning the ZGC algorithm by offering an *interactive experience*, and users can deliberately break something and observe the effects of doing so—deterministically in a fast feedback loop, in contrast to nondeterministic crashes in C++ code. Second, it allows us to *describe concisely* the most significant pieces of ZGC while *leaving out many implementation details*, providing a much smoother learning curve compared with reading the actual source code. Third, a small model facilitates *exploring and prototyping optimizations and extensions* to the ZGC algorithm *at a low cost*.

SPIN [Holzmann 2011] is a model checker for verifying the correctness of concurrent software models, which are described as a form of non-deterministic automata in the Promela language.

[28] The name is a misnomer; `jphantom` would have been more accurate to match its strength.
[29] The HotSpot JVM internally has other phantom-strength references with accessible referents; all are created using `OopStorageSet::create_weak`. Such kinds of references are used to track certain objects without keeping them unnecessarily live.
[30] The actual method used is called `NewLocalRef` or `NewGlobalRef`. Since we care only about their `get()`-related semantics in this context, we continue with the name `get()` for consistency.

Properties of interest are specified as assertions, and SPIN will explore the complete state space trying to find a state that violates the assertion and report the execution trace (the steps required to reach this problematic state). If no such violation is found, the properties are proved satisfied for this model. Since SPIN performs exhaustive searching, it does not work with models with infinite states. However, it is effective and valuable to find concurrent issues that are hard to tackle via traditional debugging techniques [Ugawa et al. 2018].

We construct a model of ZGC in Promela (Section 6.1),[31] express five interesting invariants in ZGC by way of model assertions (Section 6.2), and define some scenarios that we use to check that these invariants do hold.

6.1 Basic Entities in the Model

The two main data structures in our model are **Ordinary Object Pointer** (***OOP***) and *Obj* (for Java Object). An OOP consists of a color and an object address. An Obj mainly consists of its fields and metadata for GC, e.g., its marking status. All OOPs are stored in an array, and each OOP is uniquely identified by its index in this array. The same goes for objects as well.

Some key types are shown in Figure 22(c), such as color (M0/1, R and the finalizable-M0/1), mark status (unmarked, finalizably-marked, and strongly-marked), and ZGC phases. The names of ZGC phases are taken from the ZGC C++ source code; they correspond to the M/R, RP+EC, and RE phases used in Figure 7(a).

Finally, there are three kinds of threads in total in the model:

GC thread. A single GC thread runs *TotalGCCycles* number of complete cycles of the ZGC algorithm, from STW1 to the RE phase (pictorially shown in Figure 7(a)), where *TotalGCCycles* is a macro-defined constant. We tested up to 3 GC cycles in our runs. The skeleton model code for GC threads is shown in Figure 22(a). Lines 5–17 cover the logic of STW1 (Section 3.2), setting the good color and populating the mark stack with roots, and so on. Lines 18–19 consume/populate the mark stack while traversing the heap (Section 3.3). Lines 20–29 capture the fact that attempts to enter STW2 may fail due to concurrent push/pop of the mark stack and therefore multiple attempts may be needed (Section 3.4). Once STW2 is entered, we disable resurrection (Section 5.6). Line 32 refines the discovered list and processes each reference on the list (Section 5.7). Line 35 constructs the evacuation candidates set Section 3.6 used in the RE phase. Lines 39–40 show the good color being changed for the second time in the same ZGC cycle, and all roots are relocated if they point into the EC. Line 42 relocates all objects in EC and concludes the current ZGC cycle. Finally, Lines 43–47 check if the simulation should run another GC cycle or be terminated.

Mutator threads. The mutator logic is shown in Figure 22(b): its behavior is very limited: responding to STW pause requests (Line 11), loading reachable OOPs (Lines 17–19),[32] and no-op (Line 20). The number of mutators is controlled by a macro-defined constant, and we tested up to two mutators in our runs. The mutators terminate when the simulation is over, which always corresponds to the end of a complete ZGC cycle. We revisit some of the simplifications in Section 6.4.

Init thread. The init thread constructs the object graph, signals the start of the simulation, and waits until the simulation is finished, checking assertions in the end. In all assertions, we follow this convention: objects A and B are represented by `objs[1]` and `objs[2]`, and OOPs 1 and 2 by `oops[1]` and `oops[2]`, and so on.

[31] We cover only the essential high-level concepts. The complete model is in the supplementary material and more low-level implementation details are covered in Appendix B.

[32] The object graphs we explore are rather simple, so having a predefined array of reachable OOPs as a simplification is correct. However, for more complex object graphs (e.g., nested non-strong references) with more sophisticated assertions, this simplification must be revisited.

```
active proctype zgc() {
  simulation_ready

zgc_loop:
  stw_start(1)
  // gc sequence number
  gc_seq++
  ... // resetting all marking info
  global_phase = PhaseMark
  // flip between M0/FM0 and M1/FM1
  global_good = ...
  global_finalizable_good = ...
  // call MB on roots
  for (i : 0 .. RootsN-1) {
    mark_barrier(roots[i], SMarked)
  }
  stw_end(1)
  mark_loop:
  {...}
  stw_start(2)
  if
  :: len(mark_stack) != 0 ->
     stw_end(2)
     goto mark_loop
  :: else ->
     assert(empty(mark_stack))
     global_phase = PhaseMarkCompleted
     resurrection_blocked = true
  fi
  stw_end(2)

  refine_discovered_list()
  resurrection_blocked = false

  ... // constructing EC

  stw_start(3)
  global_phase = PhaseRelocate
  global_good = R
  ... // relocate all roots in EC
  stw_end(3)
  ... // relocate all objects in EC
  if
  :: gc_seq == TotalGCCycles -> skip
  :: else -> goto zgc_loop
  fi
  simulation_done = true
}
```

(a) Main GC logic.

```
// MutatorN is #mutators in the simulation
active [MutatorN] proctype mutator() {
  simulation_ready

  byte i
  loop:
  if
  :: simulation_done -> skip
  :: else ->
     if
     :: request_stw ->
          mutators_in_stw++;
          !request_stw;
          mutators_in_stw--
     :: else ->
          if
          :: select (i : 0 .. ReachableN-1)
              byte oop_index = reachable[i]
              load_barrier_on_oop(oop_index)
          :: skip
          fi
     fi
     goto loop
  fi
  mutators_done++
}
```

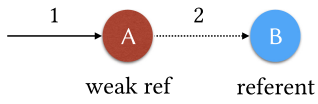
(b) Mutator logic: responding to STW pause request, loading reachable OOPs or no-op.

```
mtype:color = {FM1, FM0, R, M1, M0}
mtype:mark_status = {Unmarked, FMarked, SMarked}
mtype:phase = {PhaseMark, PhaseMarkCompleted,
               PhaseRelocate}
```

(c) Some key auxiliary types.

Fig. 22. The skeleton of (a) GC logic, (b) mutators logic, and (c) some auxiliary type definitions in the SPIN model.

The "init thread" is responsible only for initialization and checking assertions at termination, so the actual simulation involves only two parties, the GC thread and the mutators. In other words, this model highlights the interaction between the two parties in a concurrent GC context.



(a) Object graph.

```
1  if
2  :: objs[2].m == Unmarked ->
3    assert(nempty(pending_list))
4    pending_list ? ref_obj_index
5    assert(ref_obj_index == 1)
6    assert(objs[1].is_reference_type &&
7           oops[2].obj_index == 0)
8  :: else ->
9    assert(empty(pending_list))
10 fi
```

(b) Assertion after the simulation is done.

Fig. 23. Scenario 1: mutator and GC thread are racing on the reachability of the referent.

6.2 Modeling ZGC-Specific Key Invariants

The selection of features of our model is guided by our desire to capture five ZGC-specific invariants. The first three invariants focus on the interaction between GC and mutators involving non-strong references, covering the M/R, STW2, and RP phases. Invariant 1 shows that agreement on the reachability of the referent will be reached; Invariant 2 how non-strong references are picked for processing based on the reachability status; Invariant 3 demands that Invariants 1 and 2 hold simultaneously for references sharing the same referent. Given the highly concurrent nature of ZGC, the whole heap rarely holds a global property. The EC phase (between RP and STW3) is the only window where all pointers in the heap have the correct color (good or finalizable-good), as covered by Invariant 4. Invariant 5 demonstrates a subtle interaction in concurrent relocation and reclamation within the RE phase, as we alluded to in Section 3.8.

(1) GC and mutators agree on the reachability of the referent by the end of the M/R phase.
(2) A non-strong reference's eligibility for processing is determined based on the reachability of its referent. All references that are eligible for processing are processed (marked non-active and added to the *pending list*) after the RP phase.
(3) Multiple references of the same kind sharing the same referent will all receive the same treatment: all cleared or all kept intact.
(4) All OOPs have the correct color after the reference processing phase, regardless of whether they are loaded by mutators or not.
(5) No access (read/write) to reclaimed memory (objects) occurs during concurrent relocation.

6.3 Evaluation

With the invariants listed above, we would like to know whether our model upholds them, and if so, whether one can pinpoint the part of ZGC that directly corresponds to an invariant. In order to study these invariants, we construct various concrete scenarios (object graphs), and handcrafted certain assertions, reflecting the invariants. Additionally, these scenarios could serve as test cases for future research.

6.3.1 Invariant 1. To test invariant 1, we set up Scenario 1 (Figure 23) to consist of one Weak reference and its referent, connected as shown in Figure 23(a). This is enough to demonstrate the race between a mutator loading the referent and GC threads clearing the referent, the first challenge (Figure 17(a)) of concurrent reference processing. For Invariant 1 to hold, the GC and the mutator must agree on the reachability of the referent, and such judgment is final.

By the time the simulation is over, we can draw certain conclusions expressed in Figure 23(b) as assertions. If *B* is strongly marked, one can conclude that *A* is *not* in the pending list, corresponding

```
1 uintptr_t barrier(uintptr_t *slot,
2                   uintptr_t addr) {
3   // fast path
4   if (is_good_or_null(addr)) return addr;
5   // slow path
6   good_addr = ...
7   // self heal
8   self_heal(slot, addr, good_addr);
9   return good_addr;
10 }
```

(a) Fast/slow path in the barrier logic.

```
1 void self_heal(uintptr_t *slot,
2                uintptr_t old_addr
3                uintptr_t new_addr) {
4   if (new_addr == 0) return;
5   while (true) {
6     if (CAS(slot, &old_addr, new_addr))
7       return;
8     if (is_good_or_null(old_addr))
9       return;
10  }
11 }
```

(b) Self-healing to avoid hitting slow path again.

Fig. 24. Skeleton of the barrier logic and self-healing. (Reprinted from Figure 9).

to the empty pending list (Line 9); if B is not strongly-marked (because the mutator did not load Pointer 2), one can conclude B is weakly reachable and A should hence be processed (cleared) and added to the *pending list*, corresponding to assertions that the pending list is *not* empty (Line 3; `nempty` negates `empty`), the pending list contains A (Lines 4–5; ? operator extracts the first element from the list), and Pointer 2 is cleared (Line 7).

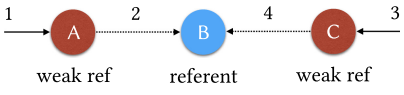
6.3.2 Invariant 2. To test Invariant 2, we reuse Scenario 1 (Figure 23) to illustrate that if a non-strong reference should be processed, it must be present in `pending_list`. Recall that we showed the main logic for self-healing without explaining the special treatment of the `NULL` value (Line 4 of Figure 24(b)). Now we can motivate its existence using SPIN. Inside the resurrection-blocked window, mutators calling `get()` to load the referent may get `NULL` as explained in Section 5.6. This prevents concurrent mutators from invalidating GC's decision on clearing a non-strong reference. In other words, the slow path of the load barrier can return `NULL`, and the pointer will be self-healed to `NULL` if the `NULL`-check was removed. (Figure 24). This may seem benign, but SPIN will report an assertion failure on Line 3 of Figure 23(b) if self-healing to `NULL` is permitted. This assertion violation means that a to-be-processed non-strong reference is missing, and it will never be placed in the notification queue (if registered).

The reason why a reference is missing is that non-strong references whose referent is `NULL` are not subject to processing. In other words, if the mutator calls `reference.clear()`, the `reference` will not participate in reference processing and therefore will *not* be marked non-active (it is already non-active from the GC's perspective) and will *not* be placed in the registered queue (the mutator will not be notified, since the mutator clears the reference itself). Self-healing to `NULL` is no different from mutators calling `clear()` from ZGC's perspective.

6.3.3 Invariant 3. To test Invariant 3, we construct Scenario 3 (Figure 25) of two Weak references pointing to the same referent, corresponding to the second challenge (Figure 17(b)) of concurrent reference processing. These two Weak references must be treated in the same manner, cleared or kept intact.

By the time the simulation is over, if B is not marked, one can conclude that the mutator did not load pointers 2 or 4, and both A and C should be processed (cleared) and added to the *pending list*. The assertion code is shown in Figure 25(b); the pending list contains two elements (Line 3) and they are objects A and C, identified by their index 1 and 3 in no particular order (Lines 5, 9, and 12).

6.3.4 Invariant 4. Scenario 4 is the same as Scenario 1 except that there is one more root that keeps the referent live. In other words, the referent is always live, regardless of whether the

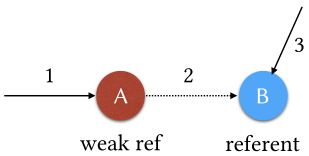


(a) Object graph.

```
1  if
2  :: objs[2].m == Unmarked ->
3      assert(len(pending_list) == 2)
4      pending_list ? ref_obj_index
5      assert(ref_obj_index == 1 || ref_obj_index == 3)
6      if
7      :: ref_obj_index == 1 ->
8          pending_list ? ref_obj_index
9          assert(ref_obj_index == 3)
10     :: ref_obj_index == 3 ->
11         pending_list ? ref_obj_index
12         assert(ref_obj_index == 1)
13     fi
14 :: else -> skip
15 fi
16 assert(empty(pending_list))
```

(b) Assertion after the simulation is done.

Fig. 25. Scenario 3: non-strong references sharing the same referent are treated uniformly.



(a) Object graph.

```
1  // assertions in the EC phase
2  // global_good will be either M0 or M1
3  assert(oops[1].c == global_good)
4  assert(oops[2].c == global_good)
5  assert(oops[3].c == global_good)
6
7  // assertions after the simulation
8  assert(objs[2].m == SMarked)
9  assert(empty(pending_list))
```

(b) Assertions in the EC phase and after the simulation.

Fig. 26. Scenario 4: all OOPs to strongly-marked objects have a good color.

mutator calls `get()` (loading the referent) or not. Therefore, *A* should not be processed (its referent is strongly-reachable) and pointer 2 should be of good color after RP.

Assertions are divided into two parts as shown in Figure 26(b). The first part claims that all OOPs have a good color, which is true only in the EC phase. The second part claims *B* is strongly marked and that the `pending_list` is empty (the non-strong reference is *not* processed).

6.3.5 Invariant 5. Recall in Section 3.8, GC threads are busy relocating all live objects on EC pages. Concurrently, a mutator that accessed an object on a page in EC could be racing with a GC thread in relocating this object, possibly creating two copies of the same object. Such races are taken care of by the CAS in the insertion operation into the forwarding table; the successful CAS defines the true destination address. However, there is also another, probably less obvious, race. Let us review the key relocation logic from the RE phase again, as shown in Figure 27.

Suppose that a mutator tries to load an object on an EC page, but does not find it in the forwarding table. Then, right before executing Line 6 in Figure 27(b), the mutator is paused. Meanwhile, GC threads, relocating all objects on EC pages, pick up this object, and successfully insert it into the forwarding table. GC threads finish relocating all live objects on this page, release this page, and pause on Line 6 in Figure 27(a). Now the mutator resumes copying the object, which would

```
1 for (page in EC) {
2   for (obj in page) {
3     relocate(obj);
4   }
5   free_page(page);
6   // page can be used for new allocation
7 }
```

(a) Main GC relocation loop.

```
1 uintptr_t relocate(uintptr_t obj) {
2   // forwarding table for this address
3   ft = forwarding_tables_get(obj);
4   if (ft->exist(obj))
5     return ft->get(obj);
6   new_obj = copy(obj)
7   // CAS inside; linearization point
8   if (ft->insert(obj, new_obj))
9     return new_obj;
10  // relocation contention
11  dealloc(new_obj);
12  return ft->get(obj);
13 }
```

(b) Thread-safe relocate function.

Fig. 27. Main GC loop in RE phase (a) and thread-safe relocation function (b) (reprinted from Figure 14).

```
1 for (page in EC) {
2   for (obj in page) {
3     relocate(obj);
4   }
5   release_page(page);
6   // depending on its RC, page may
7   // or may not be used for new allocation
8 }
```

(a) Revised main GC relocation loop.

```
1 void release_page(ZPage* page) {
2   // atomic_dec returns the old value
3   if (atomic_dec(page.rc, 1) == 1) {
4     free_page(page);
5   }
6 }
```

(b) Conditionally freeing the page depending on its RC.

Fig. 28. Revised main GC loop in RE phase (a) and RC-guarded page release (b).

access the page that has just been released, resulting in a use-after-free error [MITRE 2020]. In order to avoid this error, a handful of places need to be revised. The main idea is to manage the lifetime of each page using **reference counting** (**RC**). Figure 28 shows the revised GC loop in the RE phase and the conditional release of a page. On the mutator side, Figure 29(a) shows the actual load barrier used in the R window (Figure 8), where only Line 5 is changed to call the `relocate` wrapper, `lb_relocate`, as defined in Figure 29(b). The `lb_` prefix denotes that it is called from the load barrier. `lb_relocate` is used only by mutators because only mutators use the load barrier. Therefore, mutators guard their access to EC pages by incrementing the reference counter.

Such a use-after-free error is tricky to debug, since the freed memory may still be valid if it is yet to be returned to the OS. Therefore, we use SPIN to check that the property that all to-be-accessed pages have a positive RC is not violated. Running SPIN through the previous four scenarios shows assertion violations with the old relocation logic (Figure 27), but no failure with the revised relocation logic (Figures 28 and 29). Thus SPIN can help us find and remove subtle algorithm flaws.

6.4 Limitations

Modeling, by definition, means that certain details are omitted while focusing only on the *interesting* part. Below we document the limitations of our current model and some suggestions for future work.

(1) The mutator logic is either pointer loads or no-ops. This may at first seem like a severe restriction on what mutators can do and therefore, omits many interesting interactions between mutators and GC threads, e.g., mutators might hide objects from GC threads. However,

```
1 uintptr_t LB(uintptr_t* slot,
2              uintptr_t address) {
3   if (is_good_or_null(address)) return;
4   if (is_pointing_into(address, EC)) {
5     good_address = lb_relocate(address);
6   } else {
7     good_address = good_color(address);
8   }
9   self_heal(slot, address, good_address);
10  return good_address;
11 }
```

(a) Actual load barrier in the R window, calling the `relocate` wrapper on Line 5.

```
1 uintptr_t lb_relocate(uintptr_t obj) {
2   page = get_page(obj);
3   retained = false;
4   old_rc = page.rc
5   while (old_rc > 0) {
6     new_rc = old_rc + 1;
7     if CAS(page.rc, old_rc, new_rc) {
8       retained = true;
9       break;
10    }
11    // reloading from page.rc
12    old_rc = page.rc
13  }
14  new_obj = relocate(obj);
15  if (retained) {
16    release_page(page);
17  }
18  return new_obj;
19 }
```

(b) `relocate` wrapper to prevent the use-after-free bug.

Fig. 29. Revised load barrier in RE phase (a) and `relocate` wrapper (b).

since ZGC uses load barriers, mutators loading an object will place this object in the mark stack. Consequently, mutators cannot hide objects from GC threads, even when mutators are running concurrently with GC threads.

(2) We currently model only a single GC thread, because the most interesting cases lie in the interaction between mutators and GC threads, not among GC threads: using thread-safe `mark_obj` to support parallel marking in the M/R phase, having a per GC-thread discovered list for non-strong references in the M/R and RP phases, and assigning each to-be-relocated page to exactly one GC thread in the RE phase. Here we can see how ZGC goes to great lengths to reduce synchronization among GC threads and in the implementation, this logic is well encapsulated.

(3) In the EC phase, the per-page liveness information (the total size of live objects) is used to construct the evacuation candidates set so that only sparsely-populated pages are selected for relocation. This calculation is done by a single GC thread, so we choose not to include this logic: in our model, all live objects are added to the EC and will be relocated. Alternatively, one can utilize the nondeterminism provided by SPIN to pick a subset of relocatable pages for relocation.

7 RELATED WORK

The Compressor [Kermany and Petrank 2006] is a concurrent and parallel compaction GC algorithm implemented in Jikes RVM. It takes a standard *markbit* vector from marking, and compacts the whole heap to a single condensed area while preserving the objects' order. Internally, it utilizes two virtual spaces, both of the same size as the heap. In each GC cycle, live (determined by the *markbit* vector) objects are moved from one virtual space (*from-virtual-space*) to the other virtual space (*to-virtual-space*), and the roles of these virtual spaces alternate thereafter. Before actual compaction, an auxiliary *offset* table is constructed concurrently to help calculate the destination address for each live object, assuming all objects are copied in order. After the completion of the *offset* table, *to-virtual-space* is protected concurrently and roots are updated to point to the

corresponding addresses in *to-virtual-space* in an STW pause. From now on, mutators will never access *from-virtual-space*. After the STW pause, concurrent GC threads will copy objects from *from-virtual-space* to *to-virtual-space* according to the *offset* table. Once a page is fully populated (all objects belonging to it have been relocated), it can be unprotected. If a mutator tries to access a still-protected page, it will be trapped and perform the relocation itself. The *offset* table ensures a globally consistent relocation address for all objects in the context of concurrent and parallel compaction.

Pauseless GC [Click et al. 2005] and its generational successor, C4 [Tene et al. 2011], is a mostly concurrent, parallel, and mark-evacuate collector implemented in the Zing JVM. Pauseless GC and C4 both maintain a high-order bit (*Not-Marked-through*) in the pointers; during marking, this bit is used by a load barrier to uphold the strong tricolor invariant in the incremental-update style, in contrast to the snapshot-at-the-beginning style. During relocation, sparse pages selected for relocation will be protected, and mutators trying to access them will be trapped in the load barrier and the invalid pointer will be self-healed to the new location in to-space instead, which is similar to Baker-style relocation. Conceptually, Pauseless GC and ZGC share many similarities: embedding extra information in the higher bits of pointers, self-healing and concurrent relocation using load barriers, and so on. However, ZGC uses colored pointers consistently, covering both marking and relocation. Using colored pointers during relocation means that addresses for reclaimed objects can be reused immediately, both physical and virtual memory. In contrast, Pauseless GC and C4 can reuse only physical memory; virtual memory is reusable only after all invalid pointers are fixed. Additionally, their description of reference processing is very limited, so it is hard to properly compare with them in that regard. Finally, the source code is not accessible in the public domain, which makes it hard to use as the basis for future research.

Shenandoah [Clark et al. 2021; Flood et al. 2016] is another modern, mostly concurrent, parallel, and mark-evacuate collector in OpenJDK. During concurrent marking, it utilizes a store barrier to maintain the tricolor invariant in either snapshot-at-the-beginning or incremental-update style, configurable at JVM startup. During concurrent relocation, it uses a load barrier to intercept mutators on accessing potentially invalid pointers; if the pointer points to an object that should be relocated, the object is copied out of from-space to to-space, and the forwarding pointer is installed in the original object header using atomic instructions, and the new address is used to self-heal the original pointer. This maintains the to-space invariant that mutators never access from-space, like Baker-style relocation. Note that Shenandoah stores forwarding pointers in from-space, while ZGC uses off-heap (native) memory. Both options have pros and cons: reusing from-space memory avoids requesting extra memory from the OS but delays releasing memory occupied by from-space, since remapping all invalid pointers requires a complete heap traversal. The alternative has these two sides flipped. Another design difference is that Shenandoah does not use any bits in pointers, in contrast to the concept of colored pointers in ZGC. Partially due to this, Shenandoah supports 32-bit architectures and compressed OOPs, while ZGC works only with 64-bit architectures and uncompressed OOPs. Finally, Shenandoah processes non-strong references in a STW pause, while ZGC does it concurrently, which adds much complexity to the design and implementation (see Section 5).[33]

Sapphire [Hudson and Moss 2001; Ugawa et al. 2018] is an on-the-fly, parallel, replication copying, a collector in Jikes RVM built using MMTk [Blackburn et al. 2004b, a]. In Sapphire, a GC cycle mainly consists of four phases: Mark, Copy, Flip, and Reclaim. Phase transition is particularly

[33]Shenandoah starts to support concurrent reference processing in OpenJDK16 [Kennke 2021]. The overall algorithm is very similar to what is presented in this article: performing strongly-marking and finalizably-marking in one concurrent phase.

challenging in on-the-fly collectors, so the authors construct SPIN models to study the subtle concurrency. In contrast, ZGC uses STW pauses for such synchronization. (Currently, there are no on-the-fly collectors in OpenJDK.) In the Mark phase, incremental-update and snapshot-at-the-beginning style write barriers are used at different stages to ensure the tricolor invariant while striving for less floating garbage and faster marking termination. In the Copy phase, mutators still operate in from-space, but all updates are propagated to to-space, "semantic copying", as it is called in the article. In the Flip phase, roots are flipped one by one to to-space, and mutators start to work in to-space gradually. Since such flipping is not done in an atomic step, mutators may still access from-space, so all updates to to-space are propagated back to from-space. In the Copy and Flip phases, a load barrier is used only for pointer equality tests and reading volatile fields. Finally, in the Reclaim phase, no pointers point into memory for from-space, which can be reclaimed safely.

Ugawa et al. [2014] focused on on-the-fly non-strong reference processing and constructed SPIN models to check the correctness of the algorithm. Their work focuses on consistently clearing referents from mutators' perspective (what `Reference.get()` returns). Because final references affect only Phantom references, and `Reference.get()` unconditionally returns `NULL` for Phantom references, discussing final references is not essential to their work. In contrast, our work covers referent clearing from both the JVM's (the actual value of the `referent` field) and mutators' perspective, because when a Phantom reference is processed, its referent is cleared just like Soft/Weak references (changed since OpenJDK 9). Our explanation covers all four kinds of non-strong references, which provides the complete story of concurrent reference processing.

Hawblitzel and Petrank [2009] used the Boogie verification generator and the Z3 automated theorem prover to mechanically verify a mark-sweep collector and a Cheney copying collector. Those two collectors consist of x86 assembly language instructions and macro instructions. In order to use the theorem prover, the authors need to manually annotate the two collectors with preconditions, postconditions, invariants, and assertions. Neither collector is concurrent. Gammie et al. [2015] proved the safety of an on-the-fly mark-sweep collector under the x86 TSO memory model, using the proof assistant Isabelle/HOL. Ugawa et al. [2017] used bounded model checking to verify the copying phase of a concurrent copying collector under various memory models. Those verification works focus more on the correctness of GC algorithms. In contrast, our SPIN model, manually constructed from the ZGC source code, demonstrates some invariants in ZGC using a few small and concrete scenarios and serves as a bridge to the actual C++ codebase.

8 CONCLUSION

We hope our incremental dive, from the text and pictorial description of the overall algorithm, with and without non-strong references, to the SPIN model, capturing the key invariants and some subtle concurrency issues, flattens the learning curve of ZGC and makes the 25 KLOC C++ implementation less impenetrable. We believe that this work is valuable for building a community around ZGC (and OpenJDK), connected both to academia and industry. We hope that such documentation—both static plain English and an interactive SPIN model—provides the first step toward exploring OpenJDK with ZGC as a base for future memory management research.

APPENDICES

A VIOLIN PLOTS

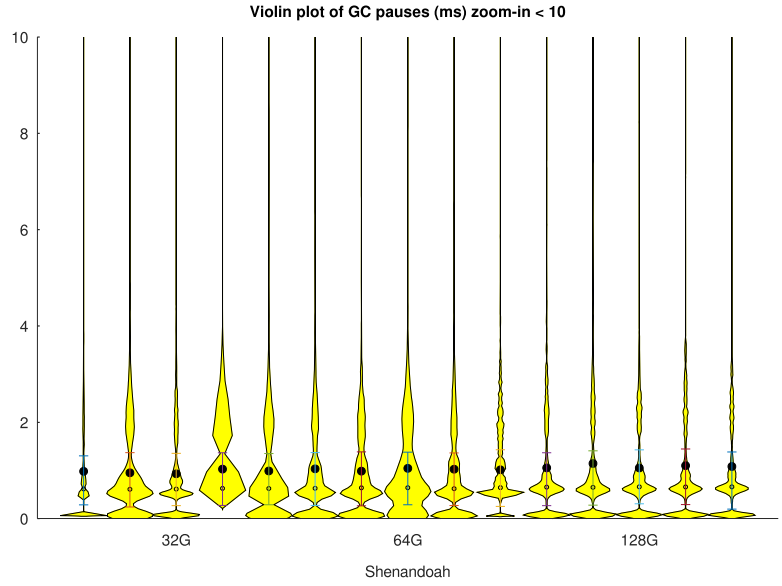


Fig. 30. Violin plots of GC pauses using Shenandoah and zoom-in < 10 ms, AMD system.

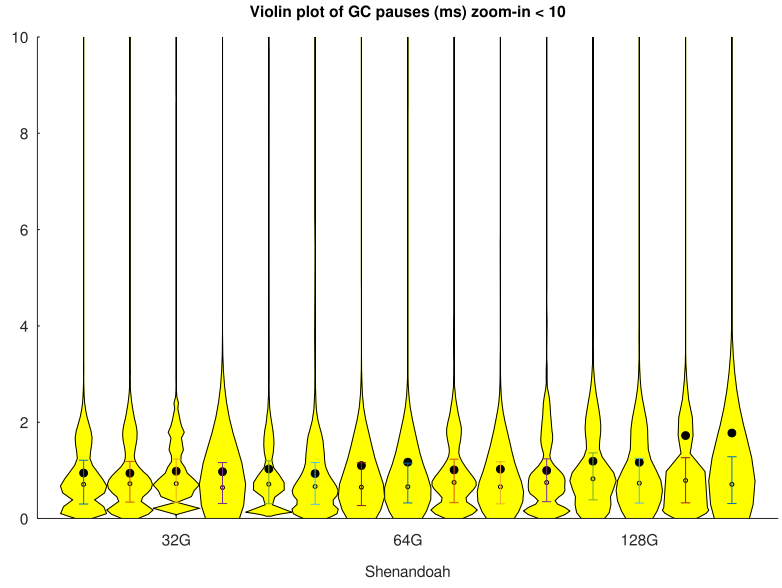


Fig. 31. Violin plots of GC pauses using Shenandoah and zoom-in < 10 ms, Intel system.

B SPIN MODEL INTERNALS

In this section, we explain some implementation details in the SPIN model and the corresponding ZGC C++ code, which hopefully helps future researchers use or extend the SPIN model.

Barrier-related code in the model (`MB`, `LB`, `barrier`, `self_heal`, and so on) is modeled from `zBarrier.inline.hpp` and `zBarrier.cpp`. We tried our best to match the actual C++ code, which results into fairly complex or odd SPIN code structures, because SPIN does not have enough expressive power, compared with C++ templates (used heavily in ZGC barrier code).

Reference processing-related code in the model (e.g., `refine_discover_list`) is modeled from `zReferenceProcessor.cpp`. In order to keep the model simple and readable, we use `chan` in SPIN to model discovered/pending lists, instead of the `discovered` field as shown in Figure 19(a).

The main ZGC loop in the model (`zgc_loop`) is modeled from `ZDriver::gc` in `zDriver.cpp`, which covers a complete GC cycle.

Since OOPs and objects are modeled separately, there is a null oop and a null object—the first element in the corresponding array, `oops[0]` and `objs[0]`, respectively, where `oops` and `objs` are the arrays holding oops and objects. In the SPIN code, for example, one can easily check if an OOP is null by testing if its index is 0.

ACKNOWLEDGMENTS

We owe a great debt of thanks to Erik Österlund, Per Lidén, and Stefan Karlsson for their thorough explanation of ZGC internals. This project would not have been possible without their assistance in navigating the complexities of ZGC's various components, such as barrier logics, colored pointers, and concurrent reference processing, among others. We are also grateful to the anonymous TOPLAS referees for their insightful and constructive remarks, which considerably enhanced this work.

REFERENCES

Henry G. Baker. 1978. List processing in real time on a serial computer. *Communications of the ACM* 21, 4 (1978), 280–294. DOI: https://doi.org/10.1145/359460.359470

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004b. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. Association for Computing Machinery, New York, NY, 25–36. DOI: https://doi.org/10.1145/1005686.1005693

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004a. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 137–146.

Roman Kennke and Christine H. Flood. 2014. JEP 189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental). Retrieved from http://openjdk.java.net/jeps/189.

Iris Clark, Roman Kennke, and Aleksey Shipilev. 2021. Shenandoah GC. Retrieved 16 may, 2020 from https://wiki.openjdk.java.net/display/shenandoah.

Cliff Click, Gil Tene, and Michael Wolf. 2005. The pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. ACM, New York, NY, 46–56. DOI: https://doi.org/10.1145/1064979.1064988

David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*. Association for Computing Machinery, New York, NY, 37–48. DOI: https://doi.org/10.1145/1029873.1029879

Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM* 21, 11 (1978), 966–975. DOI: https://doi.org/10.1145/359642.359655

Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 13:1–13:9. DOI: https://doi.org/10.1145/2972206.2972210

Peter Gammie, Antony L. Hosking, and Kai Engelhardt. 2015. Relaxing safely: Verified on-the-fly garbage collection for X86-TSO. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, 99–109. DOI: https://doi.org/10.1145/2737924.2738006

Chris Hawblitzel and Erez Petrank. 2009. Automated verification of practical garbage collectors. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, 441–453. DOI: https://doi.org/10.1145/1480881.1480935

Gerard Holzmann. 2011. *The SPIN Model Checker: Primer and Reference Manual* (1st ed.). Addison-Wesley Professional.

Richard L. Hudson and J. Eliot B. Moss. 2001. Sapphire: Copying GC without stopping the world. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*. Association for Computing Machinery, New York, NY, 48–57. DOI: https://doi.org/10.1145/376656.376810

ISO/IEC. 2011. ISO/IEC 9899:2011. Programming Language C. Retrieved 28 Dec, 2021 from https://www.iso.org/standard/57853.html.

Roman Kennke. 2019. Shenandoah GC in JDK 13, Part 1: Load Reference Barriers. Retrieved 16 April, 2021 from https://developers.redhat.com/blog/2019/06/27/shenandoah-gc-in-jdk-13-part-1-load-reference-barriers.

Roman Kennke. 2021. Shenandoah Garbage Collection in OpenJDK 16: Concurrent Reference Processing. Retrieved 16 April, 2021 from https://developers.redhat.com/articles/2021/05/20/shenandoah-garbage-collection-openjdk-16-concurrent-reference-processing.

Haim Kermany and Erez Petrank. 2006. The compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, 354–363. DOI: https://doi.org/10.1145/1133981.1134023

Per Lidén and Stefan Karlsson. 2018a. JEP 333: ZGC: A Scalable Low-Latency Garbage Collector. Retrieved 16 May, 2020 from http://openjdk.java.net/jeps/333.

Per Lidén and Stefan Karlsson. 2018b. The Z Garbage Collector—Low Latency GC for OpenJDK. Retrieved 16 May, 2020 from http://cr.openjdk.java.net/~pliden/slides/ZGC-Jfokus-2018.pdf.

MITRE. 2020. CWE-416: Use After Free. Retrieved 24 Sept., 2020 https://cwe.mitre.org/data/definitions/416.html.

Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. 2004. Mostly concurrent compaction for mark-sweep GC. In *Proceedings of the 4th International Symposium on Memory Management*. Association for Computing Machinery, New York, NY, 25–36. DOI: https://doi.org/10.1145/1029873.1029877

Pekka P. Pirinen. 1998. Barrier techniques for incremental tracing. In *Proceedings of the 1st International Symposium on Memory Management*. Association for Computing Machinery, New York, NY, 20–25. DOI: https://doi.org/10.1145/286860.286863

Ram Satish. 2016. Java WeakReference Example. Retrieved 16 May, 2020 https://www.javarticles.com/2016/10/java-weakreference-example.html. Retrived 2020-05-16.

SPEC. 2015. SPECjbb®2015. Retrieved 24 Sept, 2020 from https://www.spec.org/jbb2015/.

Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The continuously concurrent compacting collector. In *Proceedings of the International Symposium on Memory Management*. ACM, New York, NY, 79–88. DOI: https://doi.org/10.1145/1993478.1993491

Tomoharu Ugawa, Tatsuya Abe, and Toshiyuki Maeda. 2017. Model checking copy phases of concurrent copying garbage collection with various memory models. In*Proceedings of the ACM on Programming Languages*. 1, OOPSLA (2017), 26 pages. DOI: https://doi.org/10.1145/3133877

Tomoharu Ugawa, Richard E. Jones, and Carl G. Ritson. 2014. Reference object processing in on-the-fly garbage collection. In *Proceedings of the 2014 International Symposium on Memory Management*. Association for Computing Machinery, New York, NY, 59–69. DOI: https://doi.org/10.1145/2602988.2602991

Tomoharu Ugawa, Carl G. Ritson, and Richard E. Jones. 2018. Transactional sapphire: Lessons in high-performance, on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems* 40, 4 (2018), 56 pages. DOI: https://doi.org/10.1145/3226225

Paul R. Wilson. 1992. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*. Springer-Verlag, Berlin, 1–42.

Albert Mingkun Yang, Erik Österlund, Jesper Wilhelmsson, Hanna Nyblom, and Tobias Wrigstad. 2020b. ThinGC: Complete isolation with marginal overhead. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*. Association for Computing Machinery, New York, NY, 74–86. DOI: https://doi.org/10.1145/3381898.3397213

Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020a. Improving program locality in the GC using hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, 301–313. DOI: https://doi.org/10.1145/3385412.3385977

Received March 2021; revised February 2022; accepted May 2022